Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Prestor

Povzporejanje biološko navdihnjenih algoritmov

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Andrej Brodnik
Somentor: izr. prof. dr. Peter Korošec

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V praksi predstavljajo veliko skupino problemov problemi, za katere želimo dobiti vsaj približno rešitev. Dejstvo, da smo zadovoljni samo s približno rešitvijo, je posledica bodisi tega, da ne znamo najti natančne rešitve, bodisi le-ta ne obstaja. V obeh primerih se problema lotimo s tako imenovanimi metahevrističnimi pristopi. Med njimi zasedajo posebno mesto pristopi, ki so jim vzor procesi v naravi.

V diplomski nalogi preučite biološko navdihnjene metahevristike ter še posebej preučite možnost njihovega povzporejanja. Pri slednjem bodite pozorni na možnost povzporejanja na arhitekturah SIMD, katerih primer je GPGPU. Povzporejeno metahavristiko tudi implementirajte na GPGPU in ovrednotite kakovost povzporejanja.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Domen Prestor sem avtor diplomskega dela z naslovom:

Povzporejanje biološko navdihnjenih algoritmov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika in somentorstvom izr. prof. dr. Petra Korošca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 9. marca 2015 Podpis avtorja:

Zahvaljujem se mentorju, doc. dr. Andreju Brodniku, za koristne nasvete, potrpežljivost in ažurnost — za slednjo še posebej v zadnjih tednih, ko se je mudilo. Takisto se zahvaljujem izr. prof. dr. Petru Korošču za vso pomoč pri razumevanju obravnavanih tem in meritvah. Hvala tudi domačim za podporo tekom šolanja. Dovolj dolgo je trajalo, vem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ACO	ant colony optimisation	optimizacija s kolonijo mravelj
BGO	biogeographic optimisation	biogeografska optimizacija
CPU	central processing unit	centralna procesna enota
CUDA	compute unified device architecture	računsko poenotena arhitektura naprav
DASA	differential ant stigmergy algorithm	algoritem diferencialne stigmergije mravelj
DE	differential evolution	diferencialna evolucija
GA	genetic algorithm	genetski algoritem
GPU	graphics processing unit	grafična procesna enota
MIMD	multiple instruction multiple data	neodvisni ukazi in podatki
OpenCL	open computing language	odprt računski jezik
PFA	paddy field algoritem	algoritem riževih polj
PS2O	multiple particle swarm optimisation	optimizacija z več roji delcev
PSO	particle swarm optimisation	optimizacija z roji delcev
SIMD	single instruction multiple data	odvisni ukazi, neodvisni podatki

Povzetek

Diplomska naloga obravnava osnovne ideje in pristope, uporabne pri povzporejanju s strani narave navdihnjenih algoritmov. Njen temelj je poskus pohitritve algoritma, razvitega na Inštitutu Jožef Stefan, ki temelji na stigmergiji mravelj. Kako? Z uporabo grafične kartice in tovrstnemu početju namenjenega ogrodja, OpenCL. Potrebno je bilo torej pripraviti nabor tako imenovanih ščepcev kode, ki bodo delčke algoritma, primerne za vzporedno izvajanje, izvedli na grafični kartici, jih čim bolj optimizirati in opraviti potrebne meritve. Diplomsko delo opiše posamezne korake povzporejanega algoritma, poda nekaj splošnih smernic za povzporejanje biološko navdihnjenih algoritmov in predstavi konkretne rezultate meritev. Le-ti so predstavljeni v različnih kontekstih (primerjava časov izvajanj posameznih korakov algoritma glede na uporabljeno število niti, primerjava časov izvajanj posameznih korakov vzporedne implementacije s časi izvajanj posameznih korakov zaporedne in primerjava celotnega časa izvajanja vzporedne implementacije s celotnim časom izvajanja zaporedne).

Ključne besede: biološko navdihnjeni algoritmi, diferencialna stigmergija mravelj, grafična procesna enota, standard OpenCL, vmesnik JavaCL.

Abstract

This thesis addresses the basic ideas and approaches used when parallelizing biologically inspired algorithms. Its foundation is an attempt of speeding up an algorithm developed on the Jožef Stefan Institute, that is based on the stigmergy of ants. How? With the use of graphics processor and the intended framework, OpenCL. What had to be done was to prepare a set of the so called code kernels that will take pieces of the algorithm suited for parallel computing, and execute them on the graphics card; optimize them as much as possible, and perform the required measurements. The thesis describes individual steps of the parallelized algorithm, it provides some general guidelines for the parallelization of biologically inspired algorithm and presents the actual measurement results. The latter are presented in different contexts (comparison of individual algorithm steps execution times with respect to the used number of threads, comparison of individual algorithm steps parallel execution time with the individual steps sequential execution time, and comparison of the total parallel algorithm execution time with the total sequential algorithm execution time).

Keywords: biologically inspired algorithms, differential ant stigmergy, graphics processing unit, JavaCL interface, OpenCL standard.

Poglavje 1

Uvod

Narava in računalnik laiku na prvi pogled delujeta kot dva povsem različna svetova. Na eni strani je narava — kaotična, polna nepopolnosti in človeku še vedno ne povsem razumljiva. Na drugi strani pa računalnik — natančen, analitičen in predvidljiv. Oziroma takšna je ponavadi naša predstava o njiju.

V resnici računalniki niso povsem natančni in predvidljivi, kot tudi narava ne kaotična, kljub temu da se njen red pogosto ne sklada z našo idejo oziroma omejeno percepcijo reda. Rastlinski in živalski svet sta tako polna vzorcev in praviloma primitivnih (nam ne nujno intuitivnih) postopkov, razvitih skozi milijone let, s katerimi so se vrste prilagodile spremembam in tako zagotovile svoj obstoj in njega nadaljevanje — ene seveda bolj uspešno kot druge.

Pogosto uporabljen, a zato nič manj ilustrativen primer je, denimo, razvoj človeka. Začeli smo kot mikroorganizmi, šli preko rib in kuščarjev do prvih zveri in opic in trenutno smo na stopnji mislečega človeka. Vse to s ponavljanjem enih in istih korakov. To so izbor najboljših predstavnikov, reprodukcija in na koncu mutacija, ki v postopek vnese variabilnost in poskrbi za to, da se mlajše generacije razlikujejo od starejših [9].

Imamo torej tri korake, ki se ves čas izvajajo v istem zaporedju. Imamo pa tudi besedo algoritem, ki je del naslova, ampak se je do te točke še nismo dotaknili — pa bi se je počasi radi. Algoritem po definiciji ni nič drugega

kot nabor korakov in operacij, ki jih od nas zahteva rešitev določenega problema [2]. Vse to ni v računalništvu nič pretirano novega. Prvi modeli računanja, osnovani na podlagi bioloških principov, segajo vse tja v petdeseta leta prejšnjega stoletja [20]. Evolucijskim algoritmom so se kasneje pridružili še taki, ki temeljijo na podlagi rojev in kolonij, ter taki, ki temeljijo na ekologiji. Več o tem v Poglavju 2.

Tema te diplomske naloge pa kljub vsemu niso biološko navdihnjeni algoritmi sami, temveč njih povzporejanje — za kar je ena izmed poglavitnejših motivacij seveda hitrost. Določeni deli algoritma so medsebojno popolnoma neodvisni, zato ni ovir, da se ne bi izvajali sočasno. Ne nazadnje je tako tudi v naravi. Za ilustracijo se vrnimo na prej omenjeno mutacijo. Nanjo sicer lahko vplivajo razni dejavniki (le-ti so za celotno generacijo enaki in jih zato lahko odpišemo kot konstanto), organizmi pa med seboj niso odvisni in se v naravi ne čakajo, da bi videli, kaj se je zgodilo s kolegom, kot to počnejo v sekvenčnih izvedbah algoritmov.

Sočasno izvajanje je v računalništvu moč doseči na več načinov. To pot smo se odločili za uporabo grafične kartice. Naš problem je namreč pretežno podatkovno vzporeden. To v grobem pomeni, da imamo ogromno neodvisnih podatkov (kot je bilo v rahlo abstraktnejši maniri izpostavljeno že odstavek višje), denimo števil, nad katerimi želimo izvajati določeno operacijo — kar pa je pisano na kožo prav grafičnim karticam in njihovi arhitekturi SIMD. Slaba stran arhitekture SIMD je, da smo omejeni na samo eno operacijo naenkrat. To v praksi pomeni, da se razne vejitve, recimo stavek if, ne povzporejajo zelo dobro. To nas, kot bomo videli, stane, a ne toliko, da bi nas odvrnilo od uporabe grafične procesne enote.

Naslov naše teme je splošen, dočim naš pristop nekoliko manj. Povzporejanja biološko navdihnjenih algoritmov smo se lotili s povzporejanjem konkretnega biološko navdihnjenega algoritma. To nam kot zgled zadošča, ker so uporabljeni pristopi aplikativni tudi pri večini ostalih družin biološko navdihnjenih algoritmov, v določeni meri pa tudi pri splošnonamenskemu

programiranju grafičnih procesnih enot (GPGPU) obče.

Osrednji del te diplomske naloge smo razdelili na četrtine. Prva izmed njih je namenjena pregledu biološko navdihnjenih algoritmov samih. Posebej se bomo osredotočili na diferencialno stigmergijo mravelj, ki smo jo za potrebe te diplomske naloge tudi povzporejali. V Poglavju 3 si bomo nekoliko natančneje ogledali grafične kartice, njih arhitekturo, programersko paradigmo in aktualne vmesnike. Zadnja dva dela pa sta namenjena postopku povzporejanja in analizi dobljenih rezultatov meritev.

Poglavje 2

Biološko navdihnjeni algoritmi

Biološko navdihnjeni algoritmi so (poleg vsega, kar smo omenili že v uvodu) metahevristične metode za reševanje optimizacijskih problemov z uporabo stohastičnih modelov. Kar naenkrat imamo vsaj dve novi besedi, zato je prav, da nadaljujemo s kratko razlago v tem kontekstu pogosto slišanih pojmov.

- Hevristika je pristop, ki iterativno izboljšuje rešitev na podlagi dane mere kakovosti. Na tej točki je potrebno izpostaviti, da ne govorimo o popolnih rešitvah. Govorimo o rešitvah, ki se v praksi pogosto izkažejo za dovolj dobre, čeprav za to sprva nimamo nobene garancije ali pravila. Vsak hevristično pristop seveda ni dober za vsak problem.

- Za različne probleme potrebujemo različne hevristike — in to je domena metahevristike. Priljubljen internetni citat, ki ju opisuje in čigar avtor ni znan, gre v prevodu nekako takole: „Hevristika je še kar dobro pravilo. Metahevristika je še kar dobro pravilo za iskanje še kar dobrih pravil [18].“

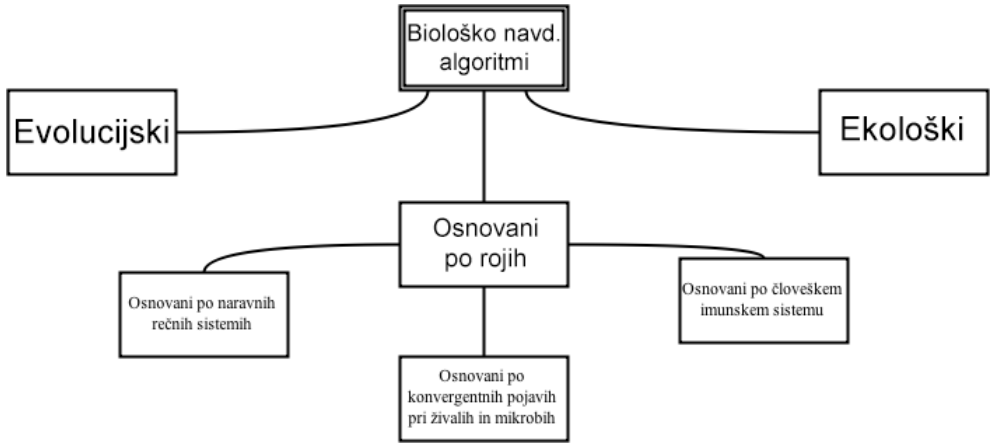
- Stohastika, na drugi strani, pa je pojem, ki opredeljuje uporabo naključnosti [29] (kot jo, denimo, lahko uporablja že omenjena mutacija).

Biološko navdihnjeni algoritmi so torej iterativni, po vzoru narave zasnovani algoritmi, ki s pomočjo naključnosti iščejo čim boljšo rešitev nekega op-

timizacijskega problema. V optimizaciji seveda poznamo tudi druge pristope — a prav biološko navdihnjeni danes veljajo za ene najučinkovitejših [22].

Problemi, ki jih rešujejo, so v praksi zelo raznoliki. NASA, recimo, je leta 2006 s pomočjo evolucijskega algoritma razvila obliko antene za vesoljska plovila. S tem so avtomatizirali prej človeško voden razvoj in pridobili na času [10] in s tem na denarju. Med zanimive probleme, s katerimi so se v preteklosti spoprijeli biološko navdihnjeni algoritmi, lahko prištejemo tudi problem tempiranja prometne signalizacije [14], finančno modeliranje [3] in druge.

2.1 Klasifikacija



Slika 2.1: Razdelitev biološko navdihnjenih algoritmov po poljih navdiha.

Slika 2.1 prikazuje klasifikacijo biološko navdihnjenih algoritmov po poljih navdiha. V resnici gre za nekoliko okleščeno verzijo drevesa, objavljenega v študiji A Survey of Bio Inspired Optimization Algorithms [5].

Kot vidimo, lahko biološko navdihnjene algoritme po teh vatlih razdelimo na tri večje družine. Na evolucijske, na ekološke in na tiste po vzoru rojev. Slednje lahko razdelimo še na tri podskupine, izmed katerih je daleč največja

(in v okviru te diplomske naloge morda tudi najbolj pomembna) srednja — socialni konvergentni pojavi pri živalih in mikrobih. V tem podpoglavju si bomo nekoliko podrobneje ogledali vsako izmed skupin, za vsako podali nekaj konkretnih primerov algoritmov in se na koncu ustavili pri algoritmu z diferencialno stigmergijo mravelj (DASA).

2.1.1 Evolucijski algoritmi

Evolucijskega algoritma (EA) smo se na kratko dotaknili že v uvodu. Kot rečeno, je ideja v osnovi dokaj preprosta. Imamo nabor posameznikov, ki ga v tem kontekstu imenujemo populacija. Poleg tega imamo pogoje — največkrat v obliki določene mere kakovosti — ki se od problema do problema razlikujejo in jim mora biti na koncu zadoščeno. Za ilustracijo si lahko predstavljamo, da ti pogoji simulirajo spremembe v okolju, katerim se najboljši posamezniki prilagodijo in najslabši podležejo. V računalništvu se dobri posamezniki od slabih ločijo po svoji uspešnosti (ang. *fitness*). Le-ta se ocenjuje glede na posameznikovo ugajanje danim pogojem. Ko enkrat vemo, kako uspešen je vsak izmed posameznikov, lahko izberemo najuspešnejše, ki bodo služili kot starši nove generacije. Zadnja koraka v iteraciji sta rekombinacija in mutacija. Prva, na podlagi dveh ali več staršev, poskrbi za kreacijo enega ali več potomcev, druga pa naključno modificira naključno izbrane dele naključno izbranih potomcev [8].

Genetski algoritem

Najbolj znan in najpogosteje uporabljen evolucijski algoritem je po Darwinovi teoriji osnovan genetski algoritem (GA). Pravzaprav lahko rečemo tudi najosnovnejši, saj zgornje definicije evolucijskega algoritma praktično ne zoži. Njemu specifično je (v okviru evolucijskih algoritmov) edino to, da posameznike imenuje kromosomi, dele posameznikov pa geni. GA je še posebej uporaben, ko je naš iskalni prostor kompleksen ali slabo definiran, ko ne vemo, kakšna je matematična analiza problema, za kompleksne ter slabo definirane

probleme in kadar tradicionalne iskalne metode odpovejo [5].

Algoritem riževih polj

Eden novejših evolucijskih algoritmov — predlagan je bil leta 2009 — je algoritem riževih polj (PFA). Algoritem temelji na razmnoževanju rastlin in ima v osnovi tri korake [30].

- Najprej imamo t. i. setev, s katero poskrbimo za inicializacijo algoritma.
- Ko rastline zrastejo, izračunamo uspešnost vsake izmed njih. Na podlagi dobljenih uspešnosti potem določimo, koliko semen bodo navrgle. Bolj uspešna rastlina ima več semen.
- Zadnji korak je oplajanje, ki poskrbi za to, da se semena prenesejo drugim rastlinam, in tako ustvarijo novo generacijo.

Diferencialna evolucija

Še en primer evolucijskega algoritma je diferencialna evolucija (DE). DE je ena izmed inačic genetskega algoritma s posebnostjo, da je njena mutacija rezultat aritmetičnih operacij nad kombinacijami različnih kromosomov. To je v kontrast zgoraj omenjenemu genetskemu algoritmu, čigar mutacija poteka nad posameznimi geni. Na ta način se mutacija spreminja in prilagaja stopnji evolucije in ni več pogojena z na začetku izbrano porazdelitvijo [5].

2.1.1.1 Primer delovanja evolucijskega algoritma

Vzemimo enostaven problem. Predstavljajmo si, da začnemo s populacijo naključnih nizov ničel in enic, radi pa bi dobili niz s samimi enicami — to bi bil naš optimum. Iz tega izhaja, da bi bila naša mera kakovosti število enic v nizu. Več kot bi imel posameznik enic, uspešnejši bi bil. Osnoven evolucijski algoritem bi do problema lahko pristopil tako, da bi izločil nize z najmanj

enicami in paroma med seboj binarno seštel najuspešnejše nize (rekombinacija). Na koncu bi lahko še stohastično izbrali bit v posamezniku, spremenili njegovo vrednost in ga s tem mutirali. Tako bi dobili novo populacijo. Ta postopek bi potem ponavljali, dokler ne bi zadostili enemu izmed izhodnih pogojev. Izhodni pogoj je lahko marsikaj. Lahko je denimo prekoračitev časovne ali iteracijske omejitve ali pa dejstvo, da je naša trenutna rešitev že dovolj dobra.

2.1.2 Ekološki algoritmi

Tudi ekologija nam je lahko zgled za računanje. V okviru nje se ne ukvarjamo z razvojem živih bitij, pač pa z njihovo interakcijo z okoljem, ki ga sestavlja bodisi živa bodisi neživa narava. Ekološki algoritmi so najmanjša izmed skupin in lahko bi rekli tudi najbolj umetno narejena. S tem mislimo predvsem to, da primerov algoritmov, ki jih bomo izpostavili v nadaljevanju, ne druži praktično nič, razen dejstva, da jih je navdihnil neki ekološki pojav (ali panoga) —, za razliko od evolucijskih, kjer smo videli, da imamo neki koren ter neko strukturo (EA), ki služi kot osnova vsem konkretizacijam (GA, PFA, DE in drugim).

Optimizacija z več roji delcev

Optimizacija z več roji delcev (PS2O) je dokaj nov algoritem, prvič predlagan leta 2008 in temelji na sočasni evoluciji in interakciji med različnimi živalskimi vrstami. Algoritem temelji na optimizaciji z roji delcev, ki jo bomo ponovno srečali v naslednjem podpoglavju. Zakaj je PSO del ene družine in PS2O druge? Kot smo že omenili, ima PS2O dimenzijo, ki je PSO nima in to je interakcija med vrstami, simbioza.

Postopek je v grobem sledeč [5].

- Na začetku ustvarimo ekosistem, ki ga sestavlja n različnih vrst, vsako vrsto pa m posameznikov. Vsak posameznik ima tudi tu svojo številko,

ki definira, kako uspešen je. Nižja kot je številka, boljši je.

- Posamezniki nato sodelujejo; najprej znotraj svoje vrste in nato še globalno in se s tem razvijajo. Uspešno sodelovanje (v obeh dimenzijah) se nagradi s padcem faktorja uspešnosti.
- Ko pridemo do situacije, kjer naša trenutna populacija ne najde več novih rešitev — kjer se je faktor uspešnosti ustalil in kjer ne pada več — to pomeni, da dani nabor bitij v danem ekosistemu ne more več (uspešno) soobstajati. To rešimo tako, da izberemo pol vrst in jih obsodimo na izumrtje, obenem pa inicializiramo prav toliko novih vrst, ki bodo od te točke naprej sodelovale s starimi vrstami, ki so preživele.

Biogeografska optimizacija

Omeniti velja tudi biogeografsko optimizacijo. Biogeografija je biološka panoga, ki se ukvarja z geografsko porazdelitvijo organizmov skozi čas. V praksi to pomeni z migracijo bitij med različnimi habitati. Pri tej paradigmi so naše rešitve različni habitati, njihova primernost za življenje pa njihova kakovost. Glavna operatorja pri BGO sta migracija in mutacija. Ideja je, da organizmi iz dobrega habitata migrirajo k slabšemu in ga tako izboljšajo. Mutacija je potrebna, ker želimo ohraniti raznolikost organizmov [5].

2.1.3 Algoritmi osnovani po rojih

V slogi je moč. Verjetno celo bolj kot mi se tega zavedajo živali, pa najsi bodo to ribe in ptice, ki se v jatah branijo plenilcev ali iščejo hrano, ali pa termiti, ki v kolonijah gradijo večmetrske termitnjake, ki jih kot posamezniki ne bi mogli. Živali, skratka, v naravi sodelujejo. To dejstvo smo srečali že pri algoritmu PS2O. Vse podobno velja tudi pri algoritmih, osnovanih po rojih, le da se ti praviloma ne ukvarjajo s simbiotskimi razmerji med vrstami. V Sliki 2.1 smo to skupino razdelili še na tri podskupine. Krajni dve bomo

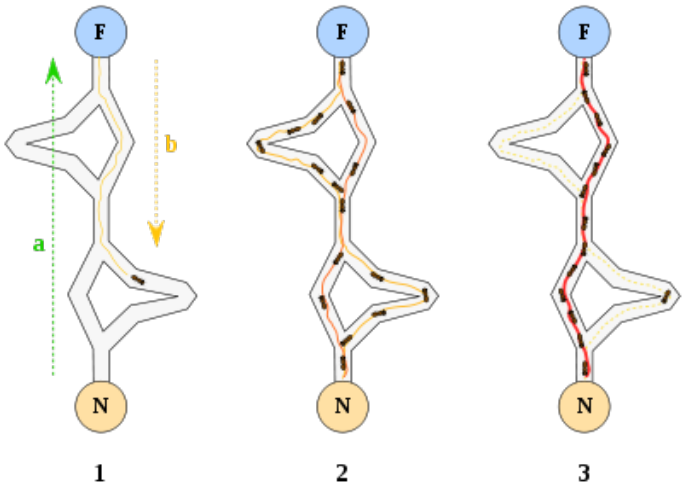
v tej diplomski nalogi zanemarili. Prav je, da vemo, da obstajata, za nas pa to pot v resnici nista preveč pomembni. Je pa res, da nekoliko razširita definicijo rojev pri „algoritmih, osnovanih po rojih“. Ena namreč temelji na naravnih rečnih sistemih, druga pa na človeškemu imunskemu sistemu (in ne na živalih, kot smo zastavili).

Podrobneje si bomo ogledali konvergentne socialne pojave pri živalih in mikrobih. Gre za algoritme, ki posnemajo obnašanje in medsebojno interakcijo pripadnikov posameznih živalskih vrst. Načinov sodelovanja je ogromno. Nekaj osnovnih idej smo podali že v začetku tega podpoglavja, pred ugotovitvijo, da roji niso nujno sestavljeni iz organizmov, nekaj pa jih bomo predelali v nadaljevanju.

Optimizacija s kolonijo mravelj

Optimizacija s kolonijo mravelj (ACO) je družina algoritmov, ki ima mnogo izpeljank. Njena omemba je v okviru tega diplomskega dela še posebej pomembna, saj je med drugim tudi osnova za algoritem DASA, ki smo ga povzporejali — vsled česar je prav, da ji namenimo nekoliko več pozornosti. Kot lahko razberemo že iz same nomenklature, ACO probleme rešuje z opazovanjem nabora mravelj. Le-te se sprehajajo po grafu in iščejo njegov globalni optimum. Pogoj je torej ta, da mora biti problem tak, da ga lahko predstavimo z grafom. V nasprotnem primeru ACO ni najprimernejša metahevristika.

Kaj pravzaprav počnejo mravlje takega, da bi nam bilo to lahko v pomoč pri računanju? Vsi smo že videli mravlje, kako hodijo v vrsti, z drobtinami kruha na hrbtu.



Slika 2.2: Prikaz iskanja optimalne poti in koordinacije s feromoni.

Da hodijo v vrsti, ni naključje. Sledijo namreč sledi feromonov, ki so jo pustile njihove predhodnice (Slika 2.2). Recimo, da F označuje mravljišče, N pa neki vir hrane. Vsaka mravlja se sprehodi od mravljišča do vira in v obe smeri pušča feromone. Če je ubrala slabo pot, bo to počela bistveno dlje časa, kot če je ubrala dobro, in posledično bo feromonov manj. Sled feromonov je od začetka šibka, zato mravlje iščejo v širino. Več kot je na neki poti feromonov, večja je možnost, da se bodo nove mravlje sprehodile po njej, in več mravelj kot se po njej sprehodi, več je feromonov. Na ta način se po nekem času oblikuje optimalna pot ali pa vsaj njen dovolj dober približek. Pojavu, kjer organizmi v naravi tako spontano sodelujejo in se med seboj koordinirajo, pravimo tudi stigmergija.

Algoritem sam bomo predstavili skozi primer. Pogost optimizacijski problem je problem trgovskega potnika. Trgovski potnik ima pred seboj zemljevid z mesti, ki jih mora obiskati, njegova želja pa je, da bi bila njegova pot čim krajša in da bi se na koncu vrnil v izhodišče. Vsako mesto si lahko ponazorimo s točko v grafu, vsako pot pa s povezavo. Optimalno je seveda, če vsako mesto obišče zgolj enkrat.

- Algoritem najprej zgradi graf, v katerem so vse točke med seboj povezane. Dolžine povezav nastavimo tako, da so proporcionalno enake dejanskim razdaljam med mesti. Vsaka pot ima dve vrednosti. Eno, ki ponazarja trenutno intenzivnost feromonov, in eno hevristično, ki je od problema do problema drugačna.

- Vsaka mravlja začne na naključno izbrani točki. Vsako naslednjo točko, ki jo bo obiskala, izbere na podlagi feromonov, ki so jih pustile ostale mravlje. Če ima mravlja na razpolago tri povezave in na nobeni ni feromonov (in hevristične vrednosti niso nastavljene), ima vsaka povezava enake možnosti, da bo izbrana. Feromoni in hevristične vrednosti delujejo kot uteži. Mravlja pomni svoje prejšnje korake, zato si med možnimi povezavami nikoli ne izbere tiste, ki vodi do mesta, ki ga je že obiskala.

- Ko je enkrat obiskala vse točke, je zgradila svojo rešitev. Te rešitve vrednotimo. Pri tem konkretnem problemu je kakovost rešitve funkcija dolžine prehojene poti.

- Ko vse mravlje zgradijo svoje rešitve, popravimo feromone. Najprej njihove vrednosti nekoliko znižamo (kot odraz dejstva, da se feromoni čez čas razpršijo), nato pa jih zvišamo glede na kvaliteto rešitev [7].

To ponavljamo, dokler ne dobimo zadovoljive rešitve.

Optimizacija z roji delcev

Optimizacija z roji delcev (PSO) temelji na obnašanju ptic, ki v jatah iščejo hrano. Delci v tem kontekstu pomenijo ptice oziroma, splošneje, enostavne agente, ki individualno gledano opravljajo razmeroma enostavne naloge. Dejansko nas zanima predvsem njihova pozicija in kako se ta, glede na njihovo hitrost in na pozicijo okolice, skozi iteracije algoritma spreminja.

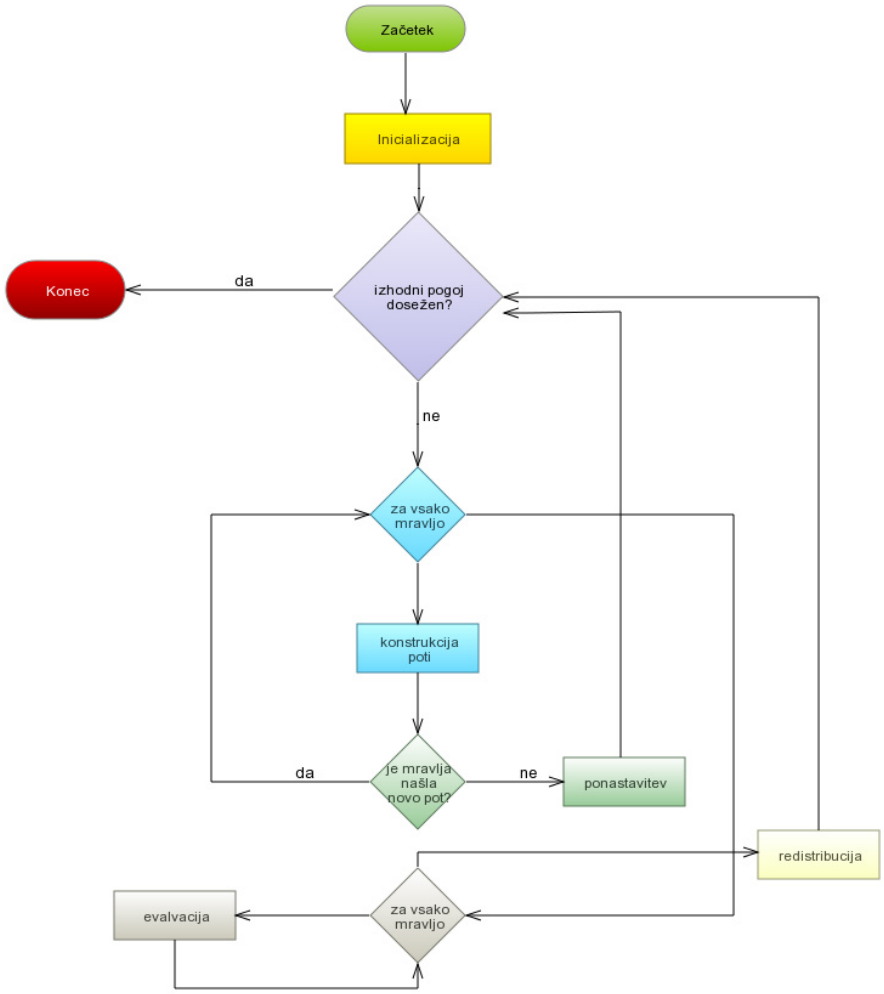
Na vsak delec v roju lahko gledamo kot na posamezno rešitev in kot na točko v večdimenzionalnem prostoru. Ta točka je predstavljena s štirimi vektorji. Eden ponazarja njegovo (delčevo) trenutno pozicijo, drugi njegovo trenutno najboljšo pozicijo (najboljšo pozicijo doslej), tretji trenutno najboljšo pozicijo njegove okolice in četrti njegovo hitrost. Tekom vsake iteracije se delci premikajo glede na svojo trenutno najboljšo pozicijo in na trenutno najboljšo pozicijo svoje okolice in na ta način sledijo najboljšim delcem [5].

2.2 Diferencialna stigmergija mravelj

Optimizacija s kolonijo mravelj sama po sebi je lahko zelo uspešna pri reševanju diskretnih optimizacijskih problemov. Če je iskalni prostor zvezen, uspešnost pade oziroma je pravilna aplikacija algoritma težja. Te omejitve so se v preteklosti lotili že večkrat in na več načinov, ampak vse rešitve so se obnesle zlasti pri problemih majhnih dimenzij. Pri večjih problemih so naletele na težave.

Na Inštitutu Jožef Stefan so leta 2012 zato predlagali algoritem DASA [15]. Citirani članek je pravi naslov za vse podrobnosti. V tem podpoglavju bomo zgolj povzeli za nas najpomembnejše koncepte. Gre za nadgradnjo klasičnega algoritma ACO, in sicer tako, da si algoritem zapomni premik (v iskalnem prostoru — grafu), ki je izboljšal trenutno najboljšo rešitev, in ga potem uporabi za osnovo nadaljnjim premikom. Najprej zgradimo diskretni graf z realnimi vrednostmi. Le-te morajo biti dokaj na gosto posejane. Ta graf je osnova za sprehod mravelj. Tak pristop algoritmu, ki je sicer namenjen reševanju diskretnih kombinatoričnih problemov, omogoča reševanje problemov v zveznem iskalnem prostoru. Težava, na katero naleti ta praksa, je, da se z rastjo števila spremenljivk poveča tudi iskalni prostor — in to za več, kot si lahko in si želimo privoščiti. Algoritem DASA ta problem rešuje tako, da si namesto dejanskih vrednosti spremenljivk zapomni samo njihove odmike, kar postopek olajša in ga naredi primernega tudi za reševanje problemov večjih

dimenzij.



Slika 2.3: Diagram poteka za algoritem DASA.

Sam algoritem se tehnično gledano bistveno ne razlikuje od algoritma ACO. Za boljšo predstavo smo pripravili (sicer zelo abstrakten) diagram poteka, ki ga prikazuje Slika 2.3 in ki nam bo, upamo, služil kot dobro izhodišče za nadaljnjo razlago.

Inicializacija

Kot prikazuje naš diagram poteka, začnemo z inicializacijo grafa. Na graf odložimo začetno količino feromonov. Tega ne storimo naključno, temveč na podlagi Cauchyjeve porazdelitvene funkcije. Le-to uporabimo, ker želimo, da so rešitve, ki so blizu trenutnemu odmiku, bolj verjetne od tistih, ki so od njega bolj oddaljene. V tem koraku izberemo tudi naključno rešitev, ki bo služila kot trenutna najboljša rešitev prvi iteraciji, in jo vrednotimo.

Konstrukcija poti

Ko je inicializacija zaključena, se vprašamo, ali je naša, naključno izbrana rešitev morda že dovolj dobra. Če ni, mravlje pričnejo z izgradnjo poti. Izgradnja poti je praktično ista kot pri algoritmu ACO. Vsaka mravlja se želi sprehoditi od začetne do končne točke. Vsak naslednji korak določi na podlagi feromonov na sosednjih točkah. Kot prikazuje Slika 2.3, imamo torej eno zanko (recimo `for`), ki skrbi za to, da se po grafu sprehodijo vse mravlje. Okvirček „konstrukcija poti“ v našem diagramu poteka pa v praksi ravno tako pomeni zanko — ki skrbi za to, da se vsaka mravlja sprehodi po celem grafu. Celoten postopek izgradnje poti se torej odvije znotraj dveh gnezdenih zank. Če v nekem predhodno določenem številu poskusov mravlja ne najde nove poti, postopek resetiramo.

Vsaka mravlja ima dve strukturi; svojo pot in svojo rešitev. Kako sestavi pot, smo že povedali. Na podlagi poti se potem izračuna odmik, ta pa se na koncu prišteje trenutni najboljši rešitvi. Mravlja tako ustvari novo rešitev, za katero pa še ne vemo, ali je boljša ali slabša od prejšnje.

Vrednotenje rešitev

Ko ima vsaka mravlja svojo rešitev, le-te vrednotimo (kot prikazuje Slika 2.3, spet znotraj zanke). To storimo tako, da na podlagi problema, ki ga rešujemo, izračunamo vrednosti posameznih rešitev. Primer vrednosti rešitve je denimo vsota vseh točk v rešitvi. Ali pa njihov minimum. O tem neko-

liko več v Poglavju 5, kjer si bomo ogledali posamezne probleme, ki smo jih optimizirali. Ko imamo vrednosti za vsako mravljo, jih primerjamo z vrednostjo trenutne najboljše rešitve. Rešitev z manjšo vrednostjo postane nova najboljša rešitev.

Prerazporeditev

Zadnji korak, prerazporeditev, poskrbi za ponovno porazdelitev feromonov glede na pot mravlje z novo najboljšo rešitvijo. Če v dani iteraciji ne najdemo nove najboljše rešitve, se feromoni razpršijo glede na faktor evaporacije — le-ta je eden od več začetnih parametrov, ki vplivajo na delovanje algoritma, ampak jih tu nismo posebej izpostavljali. Ko zaključimo s tem korakom, se ponovno vprašamo, ali je naša trenutna rešitev že dovolj dobra. To bi lahko storili že po vrednotenju in si tako prihranili zadnjo prerazporeditev.

Poglavje 3

Splošnonamensko programiranje grafičnih procesorjev

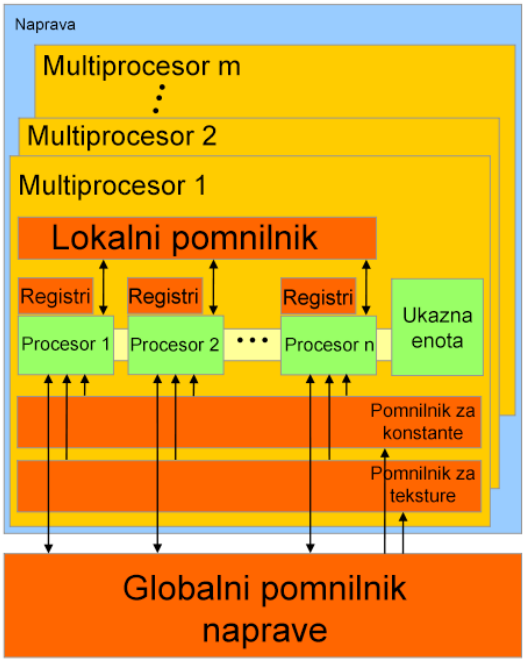
Igričarska industrija se je v zadnjih dveh desetletjih razcvetela, z njo pa je rasla tudi potreba po lepši in boljši grafiki. Lepša in boljša grafika je pogojena z boljšimi in močnejšimi grafičnimi karticami. Podjetja so bila tem pogojem pripravljena ugoditi (in tako ostati konkurenčna) in kar naenkrat — mogoče brez, da bi se tega sploh zares zavedli — smo dobili grafične procesne enote z večjo računsko močjo od tiste, ki jo premorejo centralne procesne enote. Ampak ta moč je bila mnogo let rezervirana zgolj za grafiko, za prikazovanje slik. Vse tja do leta 2007 [18], ko je Nvidia lansirala prvo verzijo svoje platforme za splošnonamensko programiranje grafičnih procesorjev, imenovane CUDA, te moči nismo znali in mogli izkoristiti. CUDA se je izkazala za odlično izhodišče, ampak žal je imela (in ima še vedno) eno hudo omejitev. Podpirajo jo izključno Nvidijine grafične kartice. Tovrsten monopol je pri konkurenci slabo čislan, zato se je kmalu zatem začelo snovanje širše alternative, ki jo dan današnji poznamo pod imenom OpenCL. Nvidijino arhitekturo bomo to pot prihranili, ker za nas in naše tokratno delo ni pomembna. Omenjamo jo

zgolj kot alternativo.

V nadaljevanju tega poglavja si bomo najprej nekoliko podrobneje ogledali arhitekturo grafičnih kartic. Poskušali bomo postaviti temelje motivom, ki so nas gnali tekom povzporejanja. Nadaljevali bomo s platformo ter standardom OpenCL in poglavje sklenili s predstavitvijo karakterističnih delov kode.

3.1 Arhitektura grafičnih procesnih enot

Če hočemo programirati grafične kartice, moramo najprej poznati njeno arhitekturo. Prinaša namreč določene omejitve in prednosti, ki korenito zaznamujejo našo programersko logiko — bolj kot jim jo pustimo zaznamovati, bolj uspešni smo lahko pri povzporejanju želenega izračuna.



Slika 3.1: Pomnilniška arhitektura grafičnih kartic.

Kot prikazuje Slika 3.1, je grafična kartica sestavljena iz globalnega pomnilnika, pomnilnika za teksture in konstante in iz več multiprocesorjev. Vsak multiprocesor je sestavljen iz lokalnega pomnilnika, ukazne enote in procesorjev; vsak procesor ima svoj nabor registrov.

- Globalni pomnilnik je posrednik, preko katerega poteka prenos podatkov med gostiteljem in napravo. Je velik, ampak počasen. Vsakršno pisanje vanj ali branje stane, zato se teh operacij tam, kjer se le da, izogibamo. Do globalnega pomnilnika — pri čemer velja izpostaviti, da lahko na oba, tako pomnilnik za konstante kot pomnilnik za teksture, gledamo kot na dela globalnega pomnilnika — lahko dostopajo vsi procesorji.

- Vsak multiprocesor ima svoj lokalni pomnilnik. Ta je hiter, a majhen. Ponavadi ga uporabimo, če do neke spremenljivke v globalnem pomnilniku dostopamo večkrat. To storimo tako, da želene podatke vanj prepišemo iz globalnega pomnilnika. Procesorji, znotraj posameznega multiprocesorja, lahko med seboj komunicirajo preko lokalnega pomnilnika.

- Registri so najmanjša izmed vseh pomnilniških struktur na GPU in so namenjeni hranjenju lokalnih virov posameznih procesorjev.

GPU ima torej veliko procesorjev, za razliko od CPU, ki jih ima dan današnji štiri, osem in več. Pri čemer velja, da ne pride niti blizu 2^8, 2^9 ali 2^{10}, kot vsak multiprocesor (na sodobni grafični kartici) vsebuje procesorjev. Procesorji na GPU so seveda šibkejši in s tem počasnejši od tistih na CPU, ampak če je želen izračun primeren za povzporejanje, njihova številčnost to odtehta.

Najpogostejši model vzporednosti, ki ga uporabljajo grafične kartice, je že omenjeni SIMD, možna pa je tudi arhitektura MIMD [19], ki je kompleksnejša za implementacijo, a potencialno učinkovitejša.

3.2 OpenCL

OpenCL je programski vmesnik, ki ga je leta 2009 [23] v odgovor Nvidijini CUDI prvič predstavila skupina podjetij za odprte standarde, imenovana *The Khronos Group*. Gre za široko platformo, ki ni namenjena zgolj programiranju grafičnih kartic, temveč tudi programiranju drugih naprav. Temelji na standardu C99.

Standard OpenCL ima svojo terminologijo. Njegov najmanjši gradnik je t. i. delovna enota. Delovne enote se povezujejo v delovne skupine. Za trenutek se vrnimo na arhitekturo GPU (Slika 3.1). Ena delovna enota ustreza (in se izvaja na) enemu procesorju, ena delovna skupina pa enemu multiprocesorju. Vsaka delovna enota (v nadaljevanju tudi nit) ima svoje registre, kjer hrani svoje lokalne spremenljivke. Vsaka delovna skupina ima svoj lokalni prostor, do katerega dostopajo vse niti znotraj nje. Potem je tu še globalni prostor, do katerega lahko dostopajo vse delovne skupine. Niti znotraj ene delovne skupine se lahko med seboj sinhronizirajo. Globalna sinhronizacija ni mogoča (drugače kot preko gostitelja).

Vsak program, ki preko vmesnika OpenCL izkorišča procesorko moč naprave (denimo grafične kartice), je sestavljen iz vsaj dveh delov. Eden se izvaja na gostitelju, eden pa na napravi. Gostiteljev del praviloma skrbi za vso potrebno inicializacijo, za podajanje ščepcev (ang. *kernel*) napravi v izvajanje in za globalno sinhronizacijo, napravin del pa za poganjanje ščepcev. Ščepec v tem kontekstu pomeni del kode, ki ga poženemo na gostitelju, izvede pa se na napravi.

3.2.1 JavaCL

JavaCL je javanski vmesnik za OpenCL in je del paketa NativeLibs4Java [21]. Podobnih vmesnikov je v resnici več. Tega smo izbrali na podlagi vrednotenja različnih javanskih vmesnikov za OpenCL (primerjali smo ga še z vmesniki Aparapi [4], Jocl [13] in LWJGL [16]), ki smo ga opravili leta 2014. Naš razlog

za izbor je bil, da smo ga ocenili kot dovolj hitrega, dovolj dobro podprtega in dovolj enostavnega za uporabo.

3.2.2 Karakteristična koda

Inicializacija OpenCL od nas vselej zahteva sledeč postopek:

- izbor želene platforme,
- izbor želene naprave,
- kreacija konteksta,
- kreacija ukazne vrste.

Platforma je konkretna implementacija OpenCL. Kot smo že izpostavili, je OpenCL širok standard, zato ima lahko neki sistem več platform. Če izberemo, denimo, platformo Intel, bomo lahko izbirali med Intelovimi napravami (CPU). Če izberemo Nvidijino, bomo lahko izbirali med Nvidijinimi napravami. Kontekst ustvarimo glede na platformo ter eno ali več naprav in ga potrebujemo za kreacijo ukazne vrste (strukture za pošiljanje ščepcev kode z gostitelja na napravo).

```
1 //kreacija programa
2 program = context.createProgram(src);
3
4 //kreacija scepca
5 calculateRandomsKernel = program.createKernel("
      calculateRandomsKernel");
```

Izsek kode 3.1: Kreacija programa in ščepca

Ko imamo ustvarjen kontekst, lahko ustvarimo program; skupek ščepcev za izvajanje na napravi (2. vrstica Kodnega izseka 3.1). V Kodnem izseku 3.1 tipa spremenljivke `src` ne vidimo — spremenljivka `src` je znakovni niz.

Dobra praksa je, da imamo vse ščepce shranjene v eni ali več datotekah (namesto da bi jih imeli zapisane znotraj programa kot znakovni niz), iz katerih beremo. Če nič drugega, si s tem olajšamo urejanje kode in jo naredimo bolj pregledno. Ko imamo ustvarjen program, lahko definiramo posamezne ščepce (5. vrstica Kodnega izseka 3.1). Argument, ki ga prejme metoda *createKernel*, je dejansko ime ščepca.

```
1 //kreacija objekta tipa Pointer
2 getSharedMemory().setPathsPointer(allocateDoubles(problem.
      getDim() * colony.getNumOfAnts()));
3
4 //inicializacija objekta Pointer s predpripravljeno tabelo
5 getSharedMemory().getPathsPointer().setArray(enarray);
6
7 //kreacija medpomnilnika z objektom tipa Pointer
8 getSharedMemory().setPathsBuffer(getSharedMemory().
      getContext().createBuffer(CLMem.Usage.InputOutput,
      getSharedMemory().getPathsPointer()));
```

Izsek kode 3.2: Kopiranje tabele z gostitelja na napravo

Če želimo prenesti podatke z gostitelja na napravo, moramo najprej ustvariti objekt tipa *Pointer* (2. vrstica Kodnega izseka 3.2). Java ne pozna kazalcev — vsaj ne tako, kot jih pozna C —, zato JavaCL to kompenzira z razredom. Za stvarjenje objekta tipa *Pointer* kličemo metodo *allocateDoubles* (oziroma *allocate-Kak-drug-podatkovni-tip*), ki ji moramo kot argument podati velikost tabele (ki jo želimo prenesti na napravo) in njen tip. Na ta način alociramo toliko bitov pomnilnika, kot je produkt velikosti tabele in velikosti tipa. V 5. vrstici Kodnega izseka 3.2 lahko danemu objektu priredimo tabelo. Ta korak ni obvezen in ga uporabimo, kadar želimo na napravo poslati tabelo z že inicializiranimi polji. Vrstica 8 je primer kreacije medpomnilnika. Tega ustvarimo na osnovi kazalca, specificirati pa moramo tudi način uporabe. V tem primeru smo ga namenili za branje in pisanje.

Ko to opravimo, je tabela napravi na razpolago (nahaja se v napravinem globalnem pomnilniku naprave).

Ko naprava konča z računanjem, lahko gostitelj do podatkov ponovno dostopa, in sicer spet preko medpomnilnika. Tu vnovič poudarjamo, da je vsakršno branje ali pisanje v globalni pomnilnik naprave časovno zelo potratno.

```
1  //nastavitev globalnih argumentov
2  super.getConstructPathsKernel().setArgs(colony.getCurrScale
       (), colony.getNumOfAnts(), colony.getProblem().getDim()
       , super.getGraphWidthBuffer(), super.getGraphBuffer(),
       super.getCurrMeansArrayBuffer(), super.
       getHighCauchyArrayBuffer(), super.
       getLowCauchyArrayBuffer(), super.getSolutionsBuffer(),
       super.getPathsBuffer(), super.getFunctionArrayBuffer(),
        super.getValuesBuffer(), super.getRandomsBuffer(),
       super.getRandoms2Buffer(), colony.getNewBestSolutionIdx
       ());
3
4  //nastavitev lokalnih argumentov
5  super.getConstructPathsKernel().setLocalArg(15, super.
       getRandomsLocalBuffer().getByteCount());
6  super.getConstructPathsKernel().setLocalArg(16, super.
       getLocalBestSolutionBuffer().getByteCount());
7
8  //zagon scepca
9  super.setPthConstrEvt(super.getConstructPathsKernel().
       enqueueNDRange(PathGetter.getQueue(), new int[]{
       MainMemory.getNumOfThreads()}, rndEvt, rnd2Evt, super.
       getClCalcEvt()));
```

Izsek kode 3.3: Zagon ščepca

Kot je razvidno iz Izseka kode 3.3, moramo ščepcu najprej nastaviti vse potrebne argumente (2. vrstica Kodnega izseka 3.3). Če gre za tabele, so ti

argumenti medpomnilniki (objekti tipa *Buffer*), ki smo jih ustvarili v kodnemu izseku 3.2, sicer pa (ničdimenzionalne) spremenljivke poljubnega tipa. JavaCL dopušča, da argumente nastavimo s klicom ene metode — vsaj, kar zadeva globalne argumente. Nastavitev lokalnih argumentov (5. in 6. vrstica Kodnega izseka 3.3) poteka nekoliko drugače, saj nas zanima samo število zlogov. Le-tega bi lahko podali kot pozitivno celo število. Mi smo namesto tega predhodno kreirali medpomnilnik in metodi *setLocalArg* podali njegovo velikost v zlogih, ki smo jo dobili s klicom metode *getByteCount*.

Ščepec poženemo s klicom metode *enqueueNDRange* (9. vrstica Kodnega izseka 3.3). Metoda vrne objekt tipa *Event* (sl. dogodek). *Event* je namenjen hranjenju informacij o izvajanju ščepca. Za zagon ščepca potrebujemo ukazno vrsto, število vseh niti, če želimo določeno velikost delovne skupine, število niti na delovno skupino in ravno tako, če želimo, objekte tipa *Event*, ki naj jih ščepec pred začetkom izvajanja počaka. Metoda sicer ni blokirajoča. Če prevajalniku izrecno ne naročimo, naj počaka, da se dogodek zaključi — to lahko storimo s klicom metode *waitFor* in sicer tam, kjer želimo —, se bo koda na gostitelju izvajala naprej.

```
1 //glava scepca
2 __kernel void prepareF1SolutionValuesKernel(int problemDim,
      int numOfAnts, __global double* values, __global
    double* solutions, __global double* functionArray,
    __local double* scratch, __local double* auxScratch)
```

Izsek kode 3.4: Primer glave ščepca

Omenili smo, da je ščepce dobro hraniti v eni ali več datotekah. Izsek kode 3.4 je iz naše .cl datoteke, v kateri hranimo ščepce. To pomeni, da ne gre več za javansko kodo, temveč za kodo po standardu OpenCL. Tabele so, kot vidimo, podane s kazalci. Predpona *__global* označuje tiste, ki so shranjene v globalnem pomnilniku, predpona *__local* pa tiste v lokalnem. Razen teh in še nekaterih dogovorov se standard OpenCL bistveno ne razlikuje od svoje

osnove, standarda C99.

Poglavje 4

Implementacija

V tem poglavju si bomo ogledali celoten postopek povzporejanja algoritma. Tekom postopka smo se učili — in se določenih stvari naučili — zato bomo izpostavili tudi določene rešitve, ki se na koncu niso izkazale za najboljše. Za začetek bomo besedo ali dve namenili izbranim orodjem in našim okvirom. Java (zaenkrat še) ni prvi jezik, ki ob omembi grafične kartice in GPGPU človeku pade na misel. Naš razlog za njeno uporabo je, da smo se še pred pričetkom ukvarjanja z diplomsko nalogo in povzporejanjem algoritmov ukvarjali z ogrodjem za modularizacijo algoritmov, imenovanim Algorithms. To ogrodje je javansko. Algoritem DASA smo najprej implementirali s tem ogrodjem, samo povzporejanje pa je v okviru tega projekta v bistvu nekakšen naslednji korak. O samem ogrodju v tem diplomskem delu ne bomo na dolgo in na široko. Mogoče samo za občutek; gre za ogrodje, ki spodbuja razbitje algoritmov na čim manjše gradnike. Ena izmed motivacij za to je potencialna možnost mešanja gradnikov med sorodnimi algoritmi, kar z drugimi besedami lahko pomeni ustvarjanje novih (mogoče boljših) algoritmov.

To omenjamo najprej zaradi Jave, ki je za seboj potegnila tudi JavaCL, ki je vmesnik za OpenCL in ki kot vsak vmesnik ima določene (kljub temu da dokaj zanemarljive) stroške. Poleg tega pa je ogrodje Algorithms vplivalo tudi na naše razmišljanje in na sam pristop do povzporejanja. Začeli smo

torej z modularno verzijo algoritma — z algoritmom, razdeljenim na neke logične korake. Ob koncu vsake iteracije ogrodje preveri končni pogoj in bodisi konča izvajanje bodisi požene nov nabor korakov. O teh korakih smo v resnici govorili že v Poglavju 2. To so konstrukcija poti, vrednotenje in ponovna porazdelitev feromonov (in ponastavitev, ki pa ni nič drugega kot nekoliko okleščena ponovitev inicializacije). Vse to v praksi pomeni, da smo imeli kodo razbito na prafaktorje in da smo jo táko tudi vzeli za izhodišče. Tekom postopka povzporejanja so, kot bomo videli, nekateri izmed korakov dobili še nekaj podkorakov — le-ti pa so v glavnem posledica omejitev, ki jih moramo pri GPGPU vzeti v zakup.

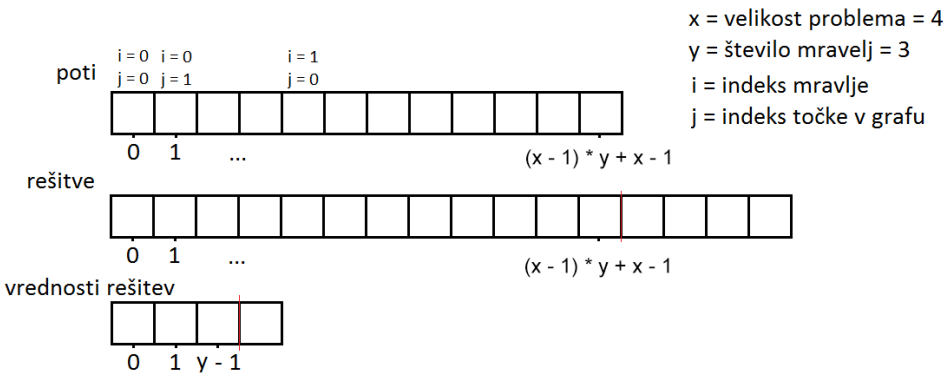
Naš pristop je bil sprva zelo naiven. Vzeli smo enega izmed gradnikov, ki smo ga ocenili kot primernega za povzporeditev, in ga dali v izvajanje na GPU. Na podlagi česa smo ga ocenili? Za začetek smo pogledali, ali ima kakšno zanko. Če da, potem smo ustvarili ščepec, ki namesto v zanki elemente dveh tabel sešteje na napravi in jih shrani v tretjo — iz nje pa potem bere gostitelj oziroma naslednji korak algoritma, ki te podatke že potrebuje. Slaba ideja. Pokazalo se je nekaj, kar smo sicer že vedeli — ampak zakaj bi zaupali teoriji? Branje in pisanje v globalni pomnilnik grafične kartice stane. In to je še toliko bolj očitno, kadar imamo opravka z večimi iteracijami, saj se stroški kopičijo. Ta zgled nam je pomagal formirati naš prvi cilj. Čim bolj zmanjšati komunikacijo med gostiteljem in napravo — obenem pa je nismo želeli povsem eliminirati, saj je bil naš motiv še vedno, da ogrodje ve za posamezne iteracije in korake algoritma, poleg tega pa tudi za njihovo uspešnost.

4.1 Inicializacija in translacija iz Jave v OpenCL

Minimizacije podatkovnega prometa med gostiteljem in napravo smo se lotili na sledeč način. Hranjenje podatkov na gostitelju očitno ni bila več možna rešitev, zato smo se odločili, da bomo takoj po inicializaciji vse pomembne

strukture prestavili na napravo in jih tam hranili do konca izvajanja algoritma.

Težava, s katero smo se soočili, je bila, da je ogrodje Algorithms javansko in s tem tudi objektno orientirano. OpenCL, na drugi strani, pa je zasnovan na standardu C99, ki ni namenjen objektno orientiranemu programiranju. Če smo si na gostitelju lahko privoščili kolonijo mravelj, definirano kot tabelo objektov tipa ***Ant***, na napravi nimamo tega luksuza. Zato smo se morali najprej odločiti, kako točno bo en objekt tipa ***Ant*** na napravi sploh predstavljen. Vsaka mravlja ima svojo pot, ki jo zgradi med vsako iteracijo. Poleg tega ima svojo rešitev, rešitev pa ima svojo vrednost. Pot in rešitev sta definirani s tabelama realnih števil (velikost obeh je enaka dimenziji problema), vrednost rešitve pa je realno število. Glede na to, da je kolonija v svojem bistvu tabela mravelj, smo se odločili, da jo bomo na GPU predstavili s tremi tabelami. Z dvema dvodimenzionalnima (rešitve in poti) in z eno enodimenzionalno (vrednosti rešitev). OpenCL v resnici ne dopušča globalnih dvodimenzionalnih tabel, zato smo ju morali simulirati z dvema enodimenzionalnima tabelama velikosti $x \cdot y$, kjer velja, da je x enak številu mravelj, y pa dimenziji problema. Iz tega velja, da je $i \cdot x$ (kjer je i indeks trenutne mravlje) enak začetku poti ali rešitve trenutne mravlje.



Slika 4.1: Predstavitev kolonije mravelj na napravi.

Za lažje razumevanje in da se ne bomo lomili z besedami, smo za ilustracijo pripravili Sliko 4.1, ki prikazuje vse, o čemer smo do te točke govorili v tem podpoglavju. Torej, če povzamemo; na Sliki 4.1 imamo kolonijo s 3 mravljami (y = 3) in problem dimenzije 4 (x = 4). Ničta mravlja ima tako v tabeli poti in rešitev rezervirana polja od 0 do vključno 3. Prva od 4 do vključno 7. Mravlja z indeksom i ima rezervirana polja od $i \cdot x$ do $i \cdot x + x - 1$. Zadnja (tretja) mravlja ima rezervirana polja od $(3-1) \cdot 4$ do $(3-1) \cdot 4 + 4 - 1$. Vsaka mravlja ima svojo rešitev in svojo pot, opredeljuje pa jo indeks (i), s pomočjo katerega lahko dostopamo do obeh. Na Sliki 4.1 lahko pri tabeli rešitev in pri tabeli vrednosti opazimo, da sta večji, kot bi glede na dano število mravelj in velikost problema morali biti. Zadnja polja v tabelah so namenjena trenutni najboljši rešitvi in vrednosti trenutne najboljše rešitve. Če je trenutna iteracija našla novo najboljšo rešitev, se ta shrani v zadnja polja tabele rešitev. Sicer zadnja polja ostanejo nespremenjena, trenutne najboljše vrednosti pa se tako prenesejo v naslednjo iteracijo.

Ničdimenzionalne spremenljivke niso problematične in jih lahko na grafično kartico ob vsaki iteraciji pošiljamo kot „navadne“ argumente ščepca. To so

razni parametri in denimo spremenljivka, ki definira obseg iskanja in ki jo iterativno popravljamo na gostitelju. Razen tega večji posegi v predstavitev posameznih struktur niso bili potrebni.

4.2 Konstrukcija poti

Recimo, da imamo (še naprej) y mravelj in problem dimenzije x.

4.2.1 Izračun Cauchyjevih limit

Prvi korak algoritma DASA je (tako v zaporedni kot tudi v naši, vzporedni implementaciji) izračun Cauchyjevih limit. Izvirno je šlo za eno zanko `for`, ki je popravljala dve tabeli velikosti x.

```
1  for i < x do
2  |  valRange ← (Integer) graphWidth[i] / 2
3  |  if valRange > 0 then
4  |  |  currMeans[i] ← cauchyXPosition[i] / valRange · 4
5  |  else
6  |  |  currMeans[i] ← 0
7  |  end
8  |  highCauchy[i] ← atan((4 − currMeans[i]) / currScale)
9  |  highCauchy[i] ← atan((−4 − currMeans[i]) / currScale)
10 end
```

Psevdokoda 4.1: Zaporedno računanje Cauchyjevih limit

Kot vidimo v Psevdokodi 4.1 (8. in 9. vrtica), algoritem v tem koraku kar na dveh točkah računa trigonometrično funkcijo arkus tangens. Ta izračun je zahteven in počasen, kar nas je motiviralo, da smo zadevo poslali v izvajanje na napravo. Zanka `for` tam odpade, saj vsaka nit (teh ustvarimo optimalno x) izvede eno iteracijo. Gre za enega izmed naših najenostavnejših ščepcev.

Kot smo omenili že čisto na vrhu, v uvodu, arhitektura SIMD ne dovoli vejitev.

```
1  //s stavkom if
2  if(valRange > 0){
3          currMeans[globalIndex] = (cauchyXPosition[
               globalIndex] / (double)valRange) * 4.0;
4  } else{
5          currMeans[globalIndex] = 0.0;
6  }
7
8  //brez stavka if
9  bool factor = valRange;
10 currMeans[globalIndex] = (cauchyXPosition[globalIndex] / (
       double)valRange) * 4.0 * factor;
```

Izsek kode 4.1: Odprava stavka if

Izsek kode 4.1 prikazuje naš poskus odpravitve vejitve. Vrstice od 2 do vključno 6 prikazujejo rešitev z vejitvijo, vrstici 9 in 10 pa rešitev brez vejitve. Spremenljivka *factor* je tipa *bool*, kar pomeni, da je enaka 0, če ji priredimo vrednost 0 in 1, če ji priredimo katero koli drugo vrednost. To je točno tisto, kar smo želeli in potrebovali. Če je *valRange* večji od 0, potem je spremenljivka *factor* (s katero pomnožimo izračun v vrstici 10) enaka 1, sicer pa 0. Z odpravitvijo vejitve velike (beri; opazne) pohitritve vseeno nismo dosegli. Razloga sta dva. Prvi je, da so sporni predvsem daljši stavki if. Tako kratki, kot je ta, pohitritve ne prizadenejo preveč. Drugi pa je, da je dimenzija naših problemov, naš x, navzgor omejen s 1000. To pomeni, da je optimalno število niti za ta ščepec maksimalno 1000. To pa je, glede na zmožnosti grafične procesne enote, razmeroma malo in vejitve tudi zato niso tako usodne.

4.2.2 Psevdo-naključna števila

Algoritem za svoje delovanje potrebuje stohastične, naključne elemente. Potrebuje tako realne, med 0 in 1, kot tudi cele, med 1 in 10 (ta 10 je definirana z enim izmed začetnih parametrov, `discreteBase`). Generiranje naključnih števil na GPU ni trivialno. Ena rešitev je, da v vsaki iteraciji na GPU pošljemo dovolj na CPU zgeneriranih naključnih števil. To je točno to, čemur bi se radi izognili.

Druga možnost, ki smo se je poslužili, je generator Mersenne twister [17]. Gre za razmeroma kompleksen algoritem, ki shranjuje svoja prejšnja stanja in na podlagi njih generira nova naključna števila. Za naše potrebe se je izkazal za prekompleksnega, poleg tega pa je deloval zelo počasi, zato smo ga kmalu opustili.

Na koncu smo se odločili za Park-Millerjev generator psevdo-naključnih števil [24], ki je enostaven za implementacijo in dosega solidne rezultate.

1 $a \leftarrow 16807$
2 $b \leftarrow 2147483647$
3 $seme \leftarrow (seme \cdot a)\%b$

Psevdokoda 4.2: Park-Millerjev algoritem za izračun naključnega števila

Algoritem v osnovi deluje tako, da vzame neko naključno število (v Psevdokodi 4.2 je to spremenljivka `seme`), ga pomnoži s 16807 in ga deli po modulu z največjim številom, ki ga dovoljuje tip `integer`. Psevdokodo 4.2 si lahko predstavljamo (in je pri naši implementaciji tudi uporabljena) kot metodo, ki kot argument dobi staro `seme`, ga preračuna kot vidimo v 3. vrstici psevdokode in vrne novo seme. Če hočemo iz spremenljivke `seme` izluščiti denimo število med 1 in 10, moramo vzeti njegov ostanek pri deljenju s številom 9 in mu prišteti 1. Torej še eno deljenje. Deljenje po modulu se je, sploh pri realnih številih, izkazalo za časovno zelo potratno operacijo, zato smo se bili primorani poslužiti alternative. Če število delimo po modulu z večkratnikom števila 2, je to isto, kot če bi namesto tega vzeli zadnjih

nekaj bitov števila (natančneje $\log_2(x)$ bitov, pri čemer je x delitelj in mora biti potenca števila 2). Največje število tipa *integer* je za ena manjše od potence števila 2, zato tu ni problema.

Uporaba tovrstnega bitnega maskiranja je zares hitrejša [12]. Problem je nastal pri drugem deljenju, kjer delitelj ni (nujno) potenca števila 2. To smo rešili tako, da smo vzeli deliteljju najbližjo potenca števila 2 in rezultat potem ustrezno popravili s prištevanjem (ali odštevanjem) zadnjega bita (1 ali 0). Naša implementacija za celo število med 1 in vključno 9:

$$x = seme\&(8-1) + seme\&(2-1) + 1$$

Prvi seštevanec je celo število med 0 in 7, drugi med 0 in 1, tretji pa 1. S tem smo statistično gledano pokvarili generator naključnih števil (nekatere vrednosti so bolj verjetne od drugih — vrednost 1, denimo, je zelo malo verjetna), ampak k sreči naš algoritem ni zelo občutljiv na naključnost teh števil. Če bi bil (in nekateri so), bi se to videlo pri rešitvah problema.

Generiranje naključnih realnih števil nam je predstavljalo še večji izziv. Na koncu smo se poslužili deljenja, ki je še vedno bistveno hitrejše od deljenja z ostankom.

```
1 b ← 2147483647
2 realnoSeme = (realnoSeme+celoSeme) / b
```

Psevdokoda 4.3: Naša „dovolj dobra“ implementacija generatorja realnih števil

Kot prikazuje Psevdokoda 4.3 (v 2. vrstici), ob vsaki iteraciji algoritma seštejemo naključno celo število, ki ga imamo iz prejšnjega odstavka, in naključno realno število iz prejšnje iteracije. Dobljeno vsoto potem delimo z največjim številom tipa integer, podobno kot prej, le da nas ne zanima ostanek, temveč dejanski količnik.

Ustvarili smo torej dva ščepca — enega za generiranje realnih števil in enega za generiranje celih števil. Na začetku izvajanja algoritma ustvarimo dve tabeli naključnih števil velikosti $x \cdot y$, ki ju pošljemo na napravo. Ščepca,

ki ju kličemo z ravno tako $x \cdot y$ nitmi, ju potem iterativno popravljata glede na opisana pravila.

4.2.3 Iskanje poti

O postopku iskanja poti pri algoritmu DASA smo že govorili v Poglavju 2. Vsaka mravlja se sprehodi po grafu in poišče svojo pot skozenj.

```
1  for i < y do
2      for j < x do
3          valRange ← (int) graphWidth_j / 2 + 0.5
4          cauchyVal ← calcPPFCauchy
5          antPath = floor((cauchyVal·(double)valRange / 4) + 0.5)
6          shiftFactor = randInt
7          while
           currBestSolution[j] + shiftFactor · graph[j] · 201 + antPath] >
           problemMax do
8              if shiftFactor > 1 then
9                  shiftFactor − −
10             else
11                 shiftFactor = randInt
12                 antPath = antPath − −
13             end
14         end
15         while
           currBestSolution[j] + shiftFactor · graph[j] · 201 + antPath] <
           problemMin do
16             if shiftFactor > 1 then
17                 shiftFactor − −
18             else
19                 shiftFactor = randInt
20                 antPath = antPath + +
21             end
22         end
23         path[j]_i ← antPath
24         offset ← shiftFactor · graph[j · 201 + antPath]
25         solution[j]_i ← currBestSolution + offset
26     end
27 end
```

Psevdokoda 4.4: Zaporedno iskanje poti

Naša implementacija korak izgradnje poti povzporeja po mravljah in po točkah v grafu. Če se osredotočimo na Psevdokodo 4.4, to pomeni, da povzporedimo obe zanki `for` (1. in 2. vrstica psevdokode). Ščepec torej poganjamo z $x \cdot y$ nitmi (pri čemer velikosti delovnih skupin nastavimo na x); v optimalnih pogojih vsaka nit opravi izračun za eno točko v rešitvi ene mravlje. Pogoj, da se lahko ščepec začne izvajati, je, da sta že končala oba ščepca, ki računata naključna števila, in ščepec, ki računa Cauchyjeve limite (metoda `calcPPFCauchy` v 4. vrstici psevdokode 4.4 na podlagi predpripravljenih Cauchyjevih limit in naključnega realnega števila izračuna konkretno vrednost `cauchyVal`).

Indeks trenutne mravlje (v psevdokodi `i`) je enak indeksu delovne skupine, indeks trenutne točke (v psevdokodi `j`) pa indeksu niti znotraj te skupine (t. i. lokalnemu indeksu). Če ščepec kličemo z manj nitmi, si pomagamo z globalnim indeksom niti, ki ga znotraj zanke povečujemo za število niti in iz njega mapiramo trenutni indeks mravlje in trenutno točko v grafu ($i = \frac{globalniIndexNiti}{x}$, $j = globalniIndexNiti\%x$).

Iz nekaterih struktur, shranjenih v globalnem pomnilniku, vsaka nit bere večkrat. To pride do izraza še posebej pri dveh zankah while (4. in 12. vrstica psevdokode), ki po potrebi popravljata nov premik v grafu. Take tabele zato prepišemo v lokalni pomnilnik (ki je, kot smo dejali, manjši, a hitrejši) in vsa nadaljnja branja opravljamo iz njega.

Kot smo že izpostavili, naključnih števil ne ustvarjamo v realnem času, temveč jih ob vsaki iteraciji pripravimo v naprej — vsaki iteraciji druge zanke `for` namenimo eno celo in eno realno naključno število. Pri zankah `while` (vrstici 8 in 16) ne moremo predvideti, kolikokrat se bosta izvedli in posledično ne, koliko naključnih števil potrebujemo. Mi smo problem rešili tako, da smo si zapomnili prvo celo naključno število (tistega, katerega smo v psevdokodi uporabili v vrstici 6) in ga uporabili kot indeks naslednjega. Vsako naslednje celo naključno število se tako v tabeli naključnih celih števil nahaja na indeksu, ki je enak prejšnjemu naključnemu celemu številu. To

pomeni, da v našem primeru (kjer imamo naključna cela števila definirana samo med 1 in 9) od prve iteracije zanke `while` naprej uporabljamo samo polja med 1 in 9. To je za naključnost (spet) slabo, ampak izkaže se, da s stališča delovanja algoritma še vedno dovolj dobro.

4.3 Ovrednotenje rešitev

Ko so vse poti in rešitve sestavljene, moramo slednje vrednotiti. Korak evalvacije smo razdelili na tri dele — na izračun vrednosti rešitve, na vrednotenje vrednosti in na prepis morebitne nove najboljše rešitve. Tudi tukaj velja, da imamo y mravelj in problem dimenzije x.

4.3.1 Redukcija

Omenili smo že, da sinhronizacija med delovnimi skupinami ni mogoča. To pomeni, da pri vsakršni globalni interakciji med elementi (najsibo to seštevanje, množenje, iskanje minimuma ...) ne moremo zagotoviti, da se rezultati med seboj ne bodo povozili. Primer. Iz globalnega pomnilnika ena nit prebere vrednost, jo poviša za 1 in shrani nazaj. V tistem, ko je prebrala vrednost, jo je lahko prebrala tudi druga nit in storila enako. Če je bila vrednost v začetku enaka 1, želimo, da bo po dveh takih seštevanjih enaka 3. Pa ne bo, ker je druga nit prišla že preden je prva število povečala in ga shranila nazaj v pomnilnik. Nova vrednost bo tako 2, kar ni prav.

```
1  j ← lokalniindeksniti
2  lokalnipomnilnik ← globalnipomnilnik
3  sinhronizacija
4  odmik ← velikostdelovneskupine / 2
5  for odmik > 0 do
6      if j < odmik then
7          lokalnipomnilnik[j] ←
           lokalnipomnilnik[j] + lokalnipomnilnik[j + odmik]
8          if prejsnjiodmik%2! = 0 AND i == 0 then
9              lokalnipomnilnik[j] ←
               lokalnipomnilnik[j] + lokalnipomnilnik[2 · odmik]
10         end
11     end
12     sinhronizacija
13     odmik ← odmik / 2
14 end
```

Psevdokoda 4.5: Redukcija za seštevanje

Kar stori redukcija, je, da v zanki paroma sešteva (oziroma nad njimi izvaja kakšno drugo operacijo) posamezne vrednosti (ki se nahajajo znotraj istega lokalnega pomnilnika) in iterativno zmanjšuje `odmik`, ki je v začetku enak polovici elementov. To pomeni, da moramo vse vrednosti najprej prepisati v lokalni pomnilnik (Psevdokoda 4.5, vrstica 2). Zatem je potrebna sinhronizacija, saj moramo zagotoviti, da so pred nadaljevanjem vse vrednosti v lokalnem pomnilniku. Seštevanje posameznih vrednosti vidimo v 7. vrstici psevdokode. Po vsaki iteraciji zanke razpolovimo odmik (13. vrstica psevdokode) in poskrbimo za vnovično sinhronizacijo niti (12. vrstica psevdokode). To počnemo, dokler je odmik večji od 0.

Če imamo 10 elementov, denimo, je prvi `odmik` enak 5. Nit z indeksom 0 tako med seboj sešteje elementa z indeksoma 0 in 5, nit z indeksom 1 sešteje elementa z indeksoma 1 in 6 itd. Ko je za nami prva iteracija zanke,

imamo tako 5 vrednosti namesto desetih, ki jih v novi iteraciji seštejemo na enak način. Potem dve in na koncu 1, ki jo lahko prepišemo nazaj v globalni pomnilnik. Po vsaki iteraciji se morajo niti med seboj počakati (sinhronizirati). Na ta način lahko izračun pohitrimo za maksimalno $\log_2(x \cdot y)$.

Če smo imeli delovnih skupin več, moramo (ali moramo res, je odvisno od tega, kaj želimo) vpeljati še eno stopnjo. V tej stopnji seštejemo delne vrednosti vseh delovnih skupin — lahko na grafični kartici, znotraj ene delovne skupine, lahko pa tudi na CPU.

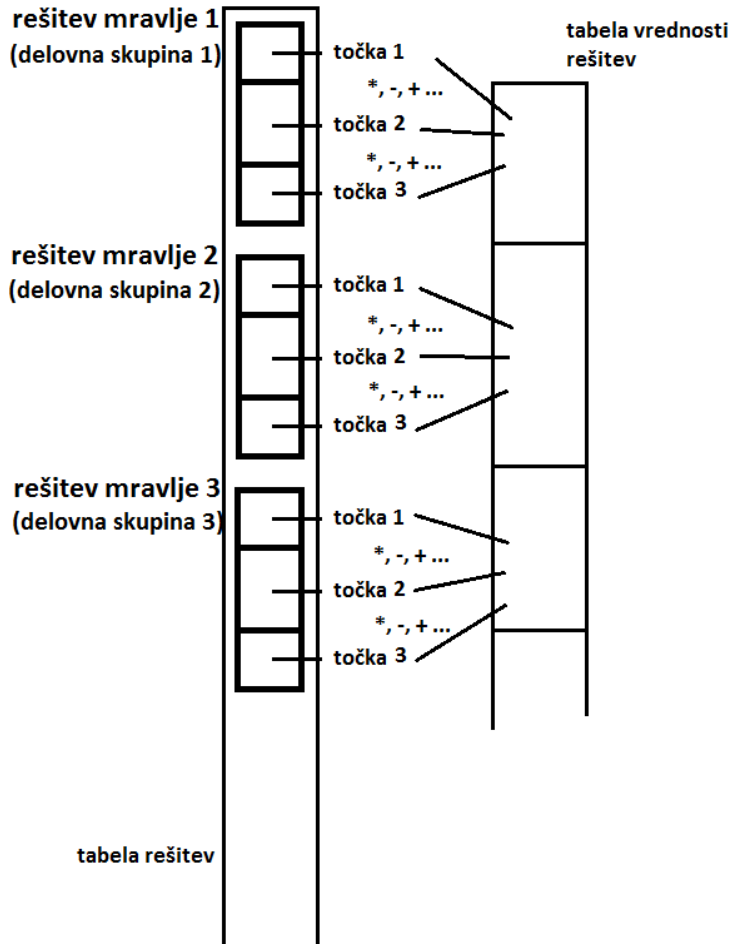
4.3.2 Priprava vrednosti rešitev

Formula, po kateri izračunamo vrednosti rešitev, zavisi od danega problema, zato smo za ta del pripravili šest različnih ščepcev kode — po enega za vsak obravnavan problem. Poganjamo jih, podobno kot iskanje poti, z $x \cdot y$ nitmi z delovnimi skupinami dimenzije x. Naša verzija tega ščepca žal ni primerna za izvajanje v drugačnih pogojih.

Delovanje ščepca je ponazorjeno na Sliki 4.2. Vsaki delovni skupini damo v izračun rešitev ene mravlje. Vsaka delovna skupina opravi redukcijo nad elementi (točkami) ene rešitve in rezultate strani na ustrezno mesto v tabeli rešitev.

Pri redukciji smo naleteli na problem. Če jo izvajamo nad tabelo elementov, katere velikost je potenca števila 2, deluje. Sicer pa pride do preskakovanja elementov. Odmik namreč razpolavljamo, in če dimenzija tabele ni potenca števila 2, slej ko prej naletimo na zaokroževanje. Vsakič, ko zaokrožimo, izpustimo en element. To je še posebej kritično proti koncu, ker je zaokroževanj več, delne vsote izgubljenih elementov pa so večje in s tem pomembnejše. Naša implementacija to težavo rešuje tako, da si vsaka iteracija redukcije zapomni prejšnji odmik — če je lih, pomeni, da namerava nova iteracija izpustiti element z indeksom $2 \cdot noviodmik$. Zato v tem primeru ta element dodamo oziroma ga odvisno od problema upoštevamo kako drugače

(Psevdokoda 4.5, vrstici 8 in 9).



Slika 4.2: Prikaz delovanja ščepca za pripravo vrednosti rešitev.

Kot vidimo na Sliki 4.2, vsaka delovna skupina računa vrednost rešitve ene mravlje (te rešitve smo v tabelo rešitev shranili v prejšnjem koraku — Psevdokoda 4.4, vrstica 25). Vrednost izračuna (na podlagi pravila, ki ga določa relevanten problem) in jo shrani v polje v tabeli vrednosti rešitev, čigar indeks je enak indeksu delovne skupine. Kot rečeno, ta implementacija ni primerna za izvajanje z manjšim številom niti od predpisanega. Če bi želeli

podpreti manjše število niti, bi morali uvesti še drugo stopnjo redukcije. To bi najverjetneje pomenilo — če bi želeli ostati v duhu trenutne implementacije (kjer se izogibamo komunikaciji med napravo gostiteljem) — še en ščepec, kjer bi denimo ena nit (oziroma več niti znotraj iste delovne skupine) opravila sekvenčni izračun.

4.3.3 Vrednotenje vrednosti rešitev

Tabela vrednosti je praviloma majhna. Za vsako mravljo imamo po eno vrednost, mravelj pa ponavadi ni več kot 50. To pomeni, da ščepec poganjamo s 50 nitmi, kar je malo. Dejansko premalo, da bi klic ščepca s časovnega vidika zares upravičili. Na tej točki smo bili v dilemi. Ali bi po izračunu vrednosti rešitev le-te poslali na CPU in vrednotenje izvedli tam ali pa bi uporabili redukcijo. Implementirali smo obe možnosti, a na koncu je obveljala druga. Izkazalo se je, da ščepec s tako malo nitmi ne odtehta časa, ki ga potrebuje za pripravo in zagon (in je dejansko počasnejši zaporedne implementacije na CPU), a nas vseeno stane manj, kot bi nas stalo pošiljanje tabel iz naprave na gostitelja.

Ta redukcija je morda nekoliko posebna tudi zato, ker je dvojna, v smislu, da na koncu hočemo tako najboljšo rešitev kot tudi njen indeks. Po zaključku redukcije 1 nit primerja najboljšo rešitev te iteracije s trenutno najboljšo rešitvijo. Če je boljša, ta nit prepiše trenutno najboljšo rešitev, jo primerja z optimumom in pripravi tabelo dveh elementov tipa integer, ki se bo poslala na CPU. To je vsa povratna informacija, ki jo ob vsaki iteraciji dobi gostitelj. Prvo polje mu pove indeks nove najboljše rešitve (če nove najboljše rešitve ni, je njegova vrednost -1), drugo pa, ali je nova rešitev že dovolj dobra (0 ali 1). To je tisti komunikacijski minimum, za katerega smo na začetku tega poglavja dejali, da ga želimo obdržati.

4.3.4 Prepis rešitve

Zadnji ščepec v tem koraku samo prepiše najboljšo rešitev v polja tabele rešitev, ki so namenjena najboljši rešitvi. Gostitelj zdaj ve, kakšen je indeks nove najboljše rešitve oziroma ali so jo mravlje v tej iteraciji sploh našle. Ta ščepec se zato pošlje v izvajanje samo takrat, ko je indeks trenutne najboljše rešitve večji od -1.

4.4 Prerazporeditev feromonov

Ta korak poskrbi za ponovno razporeditev feromonov. Izvaja se deloma na gostitelju in deloma na napravi. Če smo v dani iteraciji našli novo rešitev, potem bo gostitelj povišal globalni obseg iskanja in lokalnega nastavil na polovico globalnega. Klical se bo ščepec (idealno z x nitmi), ki bo poskrbel, da se bodo feromoni porazdelili ustrezno, glede na novo najboljšo pot. Sicer se bo globalni obseg iskanja zmanjšal, naprava pa bo poskrbela za evaporacijo feromonov (glede na ob inicializaciji algoritma opredeljeni faktor evaporacije).

4.5 Splošne ugotovitve

Tukaj bomo povzeli tiste glavne stvari, ki smo jih ugotovili (in večinoma upoštevali) tekom postopka povzporejanja.

- Komunikacija med gostiteljem in napravo je draga, vsled česar se ji velja izogniti.
- Kreacija naključnih števil na grafični kartici je že sama po sebi problem. Zavedamo se, da naša rešitev pri kakem drugem algoritmu morda ne bi delovala.

- Cena deljenja z ostankom nas je presenetila. Če se le da, se mu izognemo z uporabo bitnega maskiranja, sicer ga pač uporabimo. Pretirana telovadba s kombinacijo različnih operatorjev kot alternativo lahko premaga naš namen.
- Redukcija na GPU ni najboljša rešitev za računanje z majhnim številom elementov — kljub temu pa je težko najti boljšo.
- Človek rad spregleda, da je osnovna redukcija namenjena računanju z 2^n elementi.
- Vejitve so časovno potratne, ker se morajo zaradi arhitekture SIMD izvesti zaporedno. Kratki stavki if (in majhno število niti) pomenijo majhno časovno izgubo in jih včasih ni smiselno odpravljati.
- Dostop do globalnega pomnilnika naprave stane, zato je prav, da se izkoristi lokalnega.
- Standard OpenCL propagira širino in splošnost. Da smo se pri koraku izračuna vrednosti rešitev zadovoljili s tem, da deluje na naši napravi in dovolj dobro za nas, je slab zgled.
- Veliki ščepci so lahko zelo problematični za razhroščevanje.

Poglavje 5

Rezultati

V tem poglavju si bomo ogledali izbrane meritve, ki smo jih opravili za algoritem DASA (ki smo ga podrobno opisali v 2. in 4. poglavju). Zanimalo nas je več dimenzij merjenja (različni problemi, različne dimenzije problema, omejitev števila evalvacij, različno število mravelj ...).

5.1 Strojne specifikacije

Gostitelj, ki smo ga imeli na razpolago in s katerim smo opravljali meritve, je bil Intel i5-3570K, 3.40 gigaherčni procesor, naša naprava, na kateri smo poganjali ščepce, pa je bila Nvidijina grafična kartica GeForce GT 640, 2GB RAM-a.

5.2 Problemi

Relevantne čase smo merili za 6 problemov. Pri vseh gre za tipične optimizacijske testne funkcije (benchmark), kjer nas zanima njihov globalni minimum (v izogib izkoriščanju simetrije, zamaknjen za a). Te funkcije so sferina [28], Schwefelova [27], Rosenbrockova [26], Rastriginova [25], Griewankova [11] in Ackleyeva funkcija [1].

Kar se naše implementacije algoritma tiče, s spreminjanjem problema spreminjamo zgolj način izračuna vrednosti rešitev. V nadaljevanju te sekcije bomo podali opise posameznih načinov izračuna. Definicije funkcij so dostopne na citiranih povezavah, zato jih bomo mi raje predstavili z delčki kode, saj nam bo to prišlo prav tudi pri komentarjih rezultatov, kasneje.

Opomba; v sledečih kodnih izsekih smo zanke *for* zgolj nakazali. To smo storili zato, ker nas sam postopek redukcije tu ne zanima (predstavljen je v Sekciji 4.3.1), temveč nas zanima predvsem, kaj katera izmed funkcij računa. Nakazana zanka *for* tako v resnici ustreza zanki *for* v 5. vrstici Psevdokode 4.5.

5.2.1 Funkcija sfere

```
1  //najprej kvadriramo vse elemente
2  scratch[i] = scratch[i] * scratch[i]
3
4  //znoraj zanke for --- redukcija za vsoto vseh elementov
5  scratch[i] = scratch[i] + scratch[i + offset];
6
7  bool isntZeroThread = i;
8
9  //poskrbimo za elemente, ki izpadejo iz
10 //standardne redukcije
11 scratch[i] = scratch[i] + (prevOffset & 1) * !
       isntZeroThread * scratch[2 * offset];
```

Izsek kode 5.1: Izračun vrednosti rešitev za funkcijo sfere

Kodni izsek 5.1 prikazuje sledeče. Namesto implementirane funkcije za kvadriranje smo uporabili množenje z istim faktorjem (2. vrstica Izseka kode 5.1). Tako je namreč hitrejše. Kvadriranje morajo vse niti (znotraj delovne skupine) izvesti pred začetkom postopka redukcije. Znotraj zanke upoštevamo, da klasična redukcija izpušča elemente. Če je ostanek pri de-

ljenju prejšnjega odmika z 2 enak 1, potem moramo trenutni vsoti prve niti prišteti še element z indeksom enakim dvokratniku *odmika*. Kot prikazuje 11. vrstica Kodnega izseka 5.1, to storimo tako, da vrednosti trenutnega polja v tabeli prištejemo bodisi 0, če je prejšnji *odmik* deljiv z 2 oziroma če indeks niti ni enak 0, bodisi vrednost v polju z indeksom, enakim dvakratniku trenutnega odmika. Na ta način smo se izognili vejitvi, ki jo v 8. vrstici Psevdokode 4.5 še imamo.

5.2.2 Schwefelova funkcija

```
1  //vzamemo absolutne vrednosti vseh (razen prvega) elementov
2  //v tabeli
3  scratch[i] = fabs(solutions[antId * problemDim + i] -
       functionArray[i]);
4
5  //znoraj zanke for --- redukcija za maksimum elementov
6  bool areDifferent = fabs(scratch[i] - scratch[i + offset]);
7
8  bool thisSmaller = fabs(scratch[i] - scratch[i + offset]) -
       (scratch[i] - scratch[i + offset]) + !areDifferent;
9
10 scratch[i] = scratch[i] * !thisSmaller + scratch[i + offset
       ] * thisSmaller;
```

Izsek kode 5.2: Izračun vrednosti rešitev za Schwefelovo funkcijo

Kodni izsek 5.2 prikazuje našo implementacijo redukcijskega iskanja največje vrednosti. Tudi tokrat smo se želeli izogniti vejitvam. To smo storili s sledečim postopkom. Najprej smo se želeli prepričati, ali sta elementa enaka. Vrednost, ki jo za ta namen izračunamo, je enaka 0, če sta enaka, in 1, če nista. Nadalje preverimo, ali je trenutna vrednost manjša od odmaknjene. To storimo tako, da od absolutne vrednosti njune razlike odštejemo njuno razliko in prištejemo vrednost, ki nam pove, ali sta vrednosti enaki. Na koncu opravimo seštevanje obeh elementov, pomnoženih, enega z 0 in drugega z

1, odvisno od tega, kateri je manjši. Zaradi preglednosti smo ta del sicer izpustili, ampak tudi tu pazimo na vrednosti, ki jih redukcija preskoči.

5.2.3 Rosenbrockova funkcija

```
1 scratch[i] = 100 * (auxScratch[i] * auxScratch[i] -
      auxScratch[i + 1]) * (auxScratch[i] * auxScratch[i] -
      auxScratch[i + 1]) + (auxScratch[i] - 1) * (auxScratch[
      i] - 1);
2
3 //znoraj zanke for --- redukcija za vsoto elementov
4 scratch[i] = scratch[i] + (prevOffset & 1) * !
      isntZeroThread * scratch[2 * offset];
```

Izsek kode 5.3: Izračun vrednosti rešitev za Rosenbrockovo funkcijo

Pri Rosenbrockovi funkciji so s stališča povzporejanja zanimive predvsem prve štiri vrstice Kodnega izseka 5.3. Izračun je namreč dokaj kompleksen in kot tak idealen za izvajanje na grafični kartici.

5.2.4 Rastriginova funkcija

```
1 scratch[i] = (scratch[i] * scratch[i]) - 10 * cos(2 * M_PI
      * scratch[i]) + 10;
2
3 //znoraj zanke for --- redukcija za vsoto elementov
4 scratch[i] = scratch[i] + (prevOffset & 1) * !
      isntZeroThread * scratch[2 * offset];
```

Izsek kode 5.4: Izračun vrednosti rešitev za Rastriginovo funkcijo

Podobno kot pri Rosenbrockovi nas tudi tudi pri Rastriginovi funkciji zanima predvsem izračun v prvih dveh vrsticah Izseka kode 5.4. Našo implementacijo redukcije za vsoto elementov smo si podrobneje ogledali že pri

funkciji sfere.

5.2.5 Griewankova funkcija

```
1  scratch[i] = (scratch[i] * scratch[i]) / 4000;
2  auxScratch[i] = cos(auxScratch[i] / sqrt(i + 1.0));
3
4  //znoraj zanke for --- redukcija za
5  //in produkt elementov
6  scratch[i] = scratch[i] + scratch[i + offset];
7  auxScratch[i] = auxScratch[i] * auxScratch[i + offset];
8
9  //rezultat (po koncu redukcije)
10 values[antId] = scratch[0] - auxScratch[0] + (-179.0);
```

Izsek kode 5.5: Izračun vrednosti rešitev za Griewankovo funkcijo

Kot lahko vidimo v Kodnem izseku 5.5, je posebnost Griewankove funkcije ta, da redukcijo izvajamo nad dvema tabelama hkrati. Rezultat prve je vsota elementov, rezultat druge pa produkt. Ta posebnost s stališča povzporejanja ni zelo posebna, ampak recimo, da je to ena izmed karakteristik, ki Griewankovo funkcijo ločujejo od njenih predhodnic. Izpostaviti velja, da je računanje v drugi vrstici izseka 5.5 dokaj kompleksno.

5.2.6 Ackleyjeva funkcija

```
1  scratch[i] = scratch[i] * scratch[i];
2  auxScratch[i] = cos(2 * M_PI * auxScratch[i]);
3
4  //znoraj zanke for --- redukcija za vsoto elementov
5  scratch[i] = scratch[i] + scratch[i + offset];
6  auxScratch[i] = auxScratch[i] + auxScratch[i + offset];
7
8  //rezultat (po koncu redukcije)
9  scratch[0] = -20 * exp(-0.2 * sqrt(scratch[0] / problemDim)
       ) - exp(auxScratch[0] / problemDim) + 20 + M_E;
```

```
10  values[antId] = scratch[0] - 140;
```

Izsek kode 5.6: Izračun vrednosti rešitev za Ackleyevo funkcijo

Kodni izsek 5.6 prikazuje zelo podobno sliko kot izsek 5.5 — Ackleyjeva funkcija je s stališča povzporejanja zelo podobna Griewankovi.

5.3 Meritve in rezultati

Za začetek bomo predstavili pojma relativna in dejanska pohitritev. Relativna pohitritev je definirana kot:

$$Sr(n) = \frac{\text{vzporedni algoritem na enem procesorju vzp. računalnika}}{\text{vzporedni algoritem na n procesorjih vzporednega računalnika}}$$

Dejanska pohitritev pa kot:

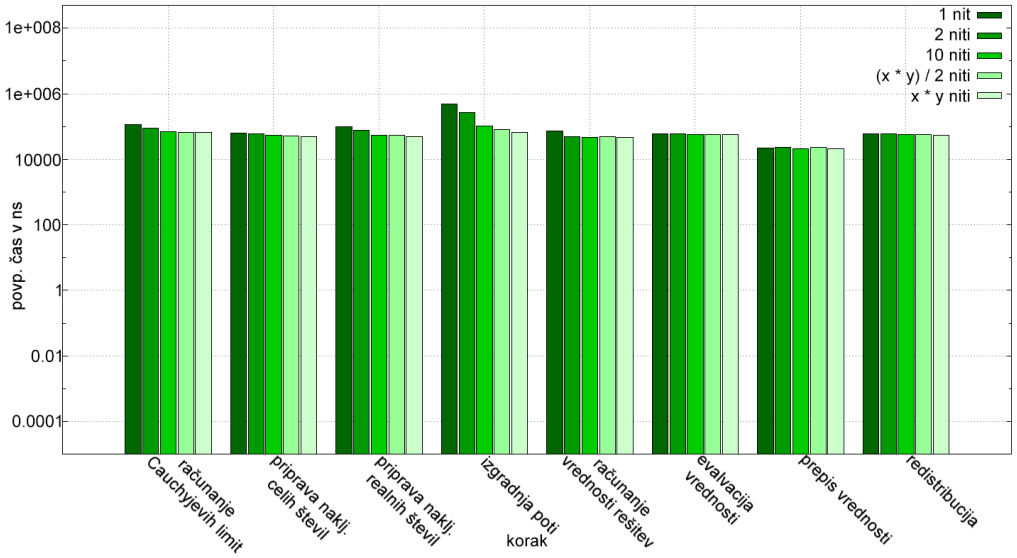
$$Sd(n) = \frac{\text{zaporedni algoritem na enem procesorju vzp. računalnika}}{\text{vzporedni algoritem na n procesorjih vzporednega računalnika}}$$

Zanimali so nas torej trije različni časi. Čas izvajanja sekvenčne implementacije na CPU. Čas izvajanja vzporedne implementacije z n nitmi. In čas izvajanja vzporedne implementacije z 1 nitjo. Preden nadaljujemo; za število niti bomo v tem poglavju uporabljali črko n. Črki x in y bosta tudi to pot dimenzija problema in število mravelj. Črka a pa bo število evalvacij.

5.3.1 Razmerje časov posameznih korakov algoritma

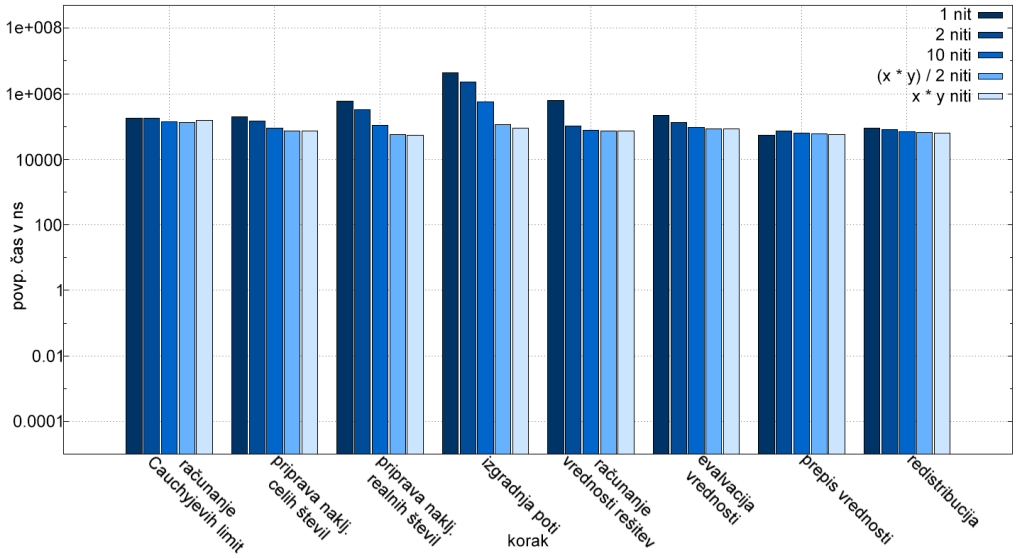
Kot smo podrobno opisali v Poglavju 4, smo svojo implementacijo razdelili na 8 korakov — 8 ščepcev. Ti koraki so računanje Cauchyjevih limit, priprava naključnih celih števil, računanje naključnih realnih števil, izgradnja poti, računanje vrednosti rešitev, evalvacija vrednosti, prepis vrednosti in redistribucija. Najbolj osnovna stvar, ki nas je zanimala, je, kako dolgo se v povprečju izvaja vsak izmed teh korakov. Meritve smo opravili za x = 10, x = 100, x = 1000; za y = 5, y = 30, y = 50 in za n = 1, n = 2, n = 10, n = $\frac{x \cdot y}{2}$, $n = x \cdot y$. Vrednost $x \cdot y$ predstavlja optimalno število niti za

izvajanje algoritma. Nekateri ščepci jih za optimalno delovanje potrebujejo manj, kar bo razvidno iz predstavljenih histogramov. Predstavili bomo le izbrane histograme — tiste, za katere ocenjujemo, da so najbolj zanimivi. Uporabili smo logaritemsko skalo. Razlog za to je, da so razlike med časi izvajanja posameznih korakov lahko zelo velike.



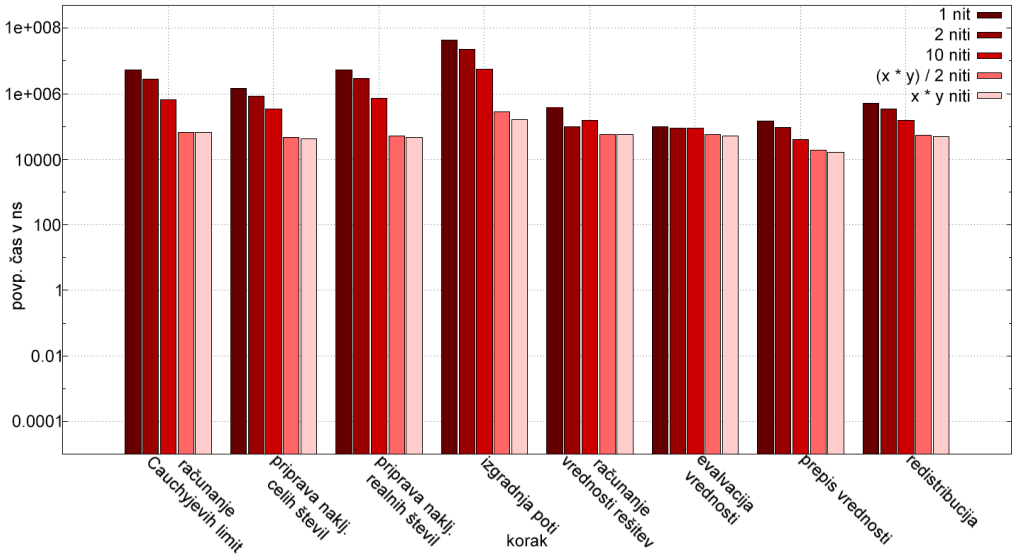
Slika 5.1: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra; x = 10, y = 5.

Histogram na Sliki 5.1 prikazuje najprej razmerje med časi izvajanj posameznih korakov (le-ti so podrobneje opisani v Poglavju 4) z različnim številom niti in nato še razmerje med časi izvajanj različnih korakov. Preden se vržemo v analizo, moramo izpostaviti, da smo ščepec, ki računa vrednosti rešitev, pognali samo z 1 nitjo in z $x \cdot y$ niti. Rezultati za 2, 10 in $\frac{x \cdot y}{2}$ niti so tako v resnici rezultati za $x \cdot y$ niti (od tod nihanje). Malo zaradi logaritemske skale (na linearni bi bile razlike bolj vidne), predvsem pa zaradi cene samega zagona ščepca (ki je dejansko višja od tiste za izvajanje ščepca) so si vsi časi zelo podobni.



Slika 5.2: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra; x = 10, y = 50.

Če povečamo zgolj število mravelj, dobimo sliko (5.2), zelo podobno prvi. Ščepci za računanje Cauchyjevih limit, evalvacijo vrednosti rešitev in za redistribucijo so opravili natanko isto delo kot pri meritvah, ki jih prikazuje Slika 5.1, zato se tudi njihovi časi izvajanja bistveno ne razlikujejo.



Slika 5.3: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra; x = 1000, y = 5.

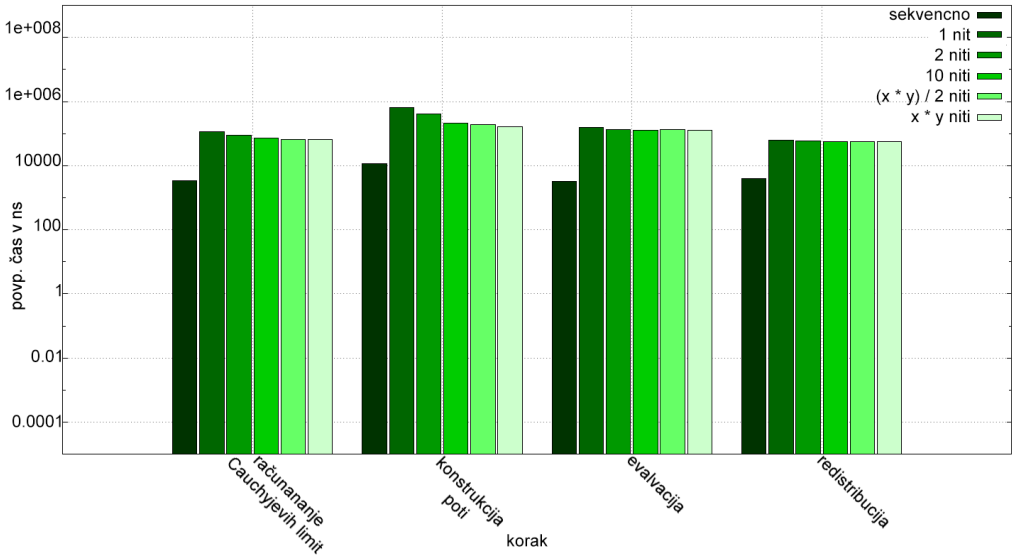
Za meritve, ki jih prikazuje Slika 5.3, smo zvišali dimenzijo problema. Tu so razlike že bolj očitne. Lepo razvidno je tudi, da nekateri ščepci ne morejo izkoristiti polnega, $x \cdot y$ števila niti. Dejansko ga lahko izkoristijo samo oba ščepca za računanje naključnih števil, ščepec za konstrukcijo poti in ščepec za računanje vrednosti rešitev.

5.3.2 Razmerje časov posameznih korakov algoritma in primerjava z njihovo sekvenčno implementacijo

Pripravili smo nabor histogramov, ki poleg časov vzporedne prikazujejo tudi čase sekvenčne (CPU) implementacije. Še vedno nas zanimajo posamezni koraki, ki pa smo jih morali za potrebe te primerjave nekoliko prirediti. Sekvenčna implementacija algoritma do stvari pristopa nekoliko drugače in nekaterih gradnikov, ki jih ima vzporedna, preprosto nima.

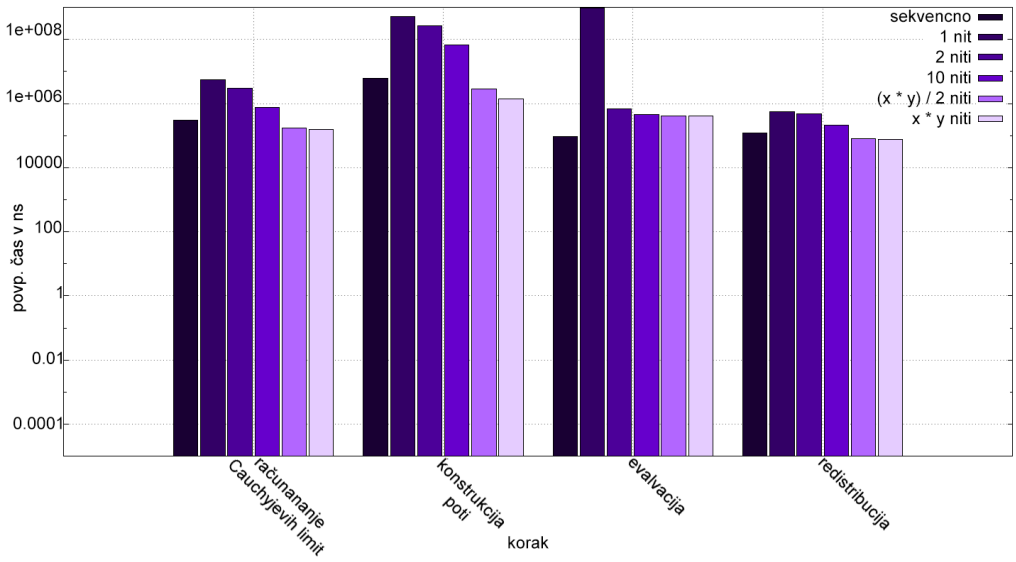
To konkretno pomeni, da smo 8 vzporednih korakov združili v 4 nove, krovne korake, ki smo jih brez težav poiskali in definirali tudi pri sekvenčni implementaciji. To so računanje Cauchyjevih limit, konstrukcija poti, evalvacija in redistribucija. Čas nove konstrukcije poti smo dobili tako, da smo sešteli časa izvajanj ščepcev za računanje obeh vrst naključnih števil in čas konstrukcije poti, čas nove evalvacije smo dobili tako, da smo sešteli čas izračuna vrednosti, čas evalvacije vrednosti in čas prepisa rešitev. Časa računanja Cauchyjevih limit in redistribucije pa sta v obeh verzijah enaka.

Opravili smo meritve za enake parametre kot prej.



Slika 5.4: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra (tudi sekvenčno); x = 10, y = 5.

Iz slike 5.4 je razvidno predvsem to, da pri tako majhnem problemu, s tako malo mravljami, z našo vzporedno implementacijo nimamo kaj iskati. Stroški zagona ščepcev in pisanja ter branja iz globalnega pomnilnika so enostavno previsoki, da bi se lahko kosali s sekvenčno implementacijo. Vrh tega pa velja, da so procesorji na grafični kartici praviloma šibkejši od tistih na centralni procesni enoti. Pri veliko nitih njihova številčnost to odtehta, pri petdesetih pa ne.



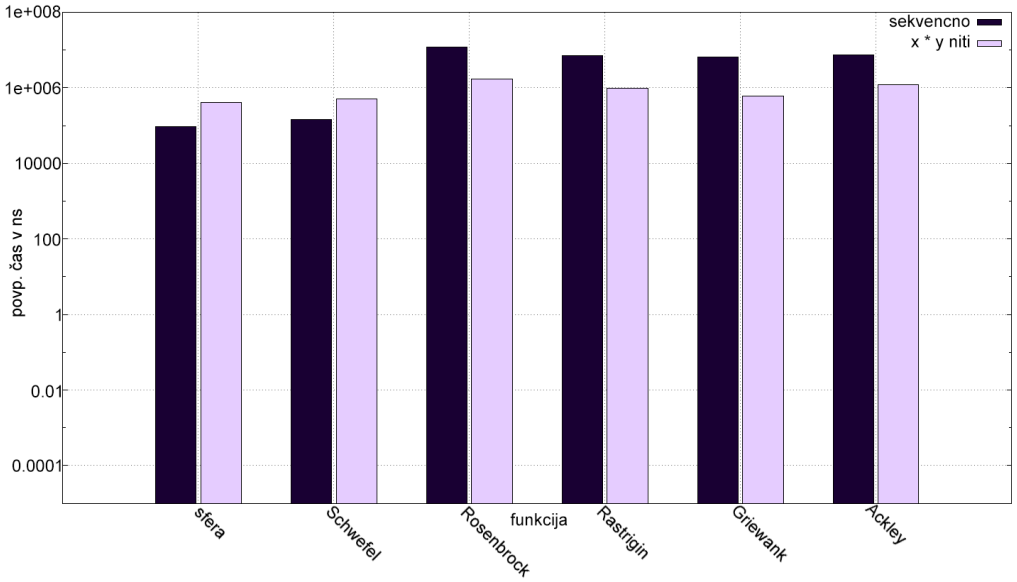
Slika 5.5: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra (tudi sekvenčno); x = 1000, y = 50.

Če maksimiziramo dimenzijo problema in število mravelj (kot prikazuje histogram na Sliki 5.5), so rezultati obetavnejši. Edini korak algoritma, kjer naša implementacija še vedno klone pred sekvenčno, je evalvacija. Razloga sta (vsaj) dva. Prvi je, da je izračun vrednosti rešitve za funkcijo sfere razmeroma enostaven, vsled česar rezultati povzporejanja niso tako dobri, kot bi bili sicer. Drugi pa je, da k času izvajanja prištevamo tudi vrednotenje vrednosti, ki se tudi v tem primeru izvaja le s 50 nitmi. Poleg tega pa je tu še tretji ščepec (prepis rešitev), ki k vsoti, če nič drugega, doprinese svoje stroške zagona.

5.3.3 Primerjava časov izvajanja koraka evalvacije novih rešitev za različne funkcije

Problem je opredeljen s ščepcem, ki računa vrednosti rešitev, oziroma obratno. To je del evalvacije, zato bomo v tem podpoglavju predstavili še histogram, ki predstavlja povprečne čase izvajanj evalvacije rešitev. Zanima nas

primerjava sekvenčne implementacije z vzporedno.



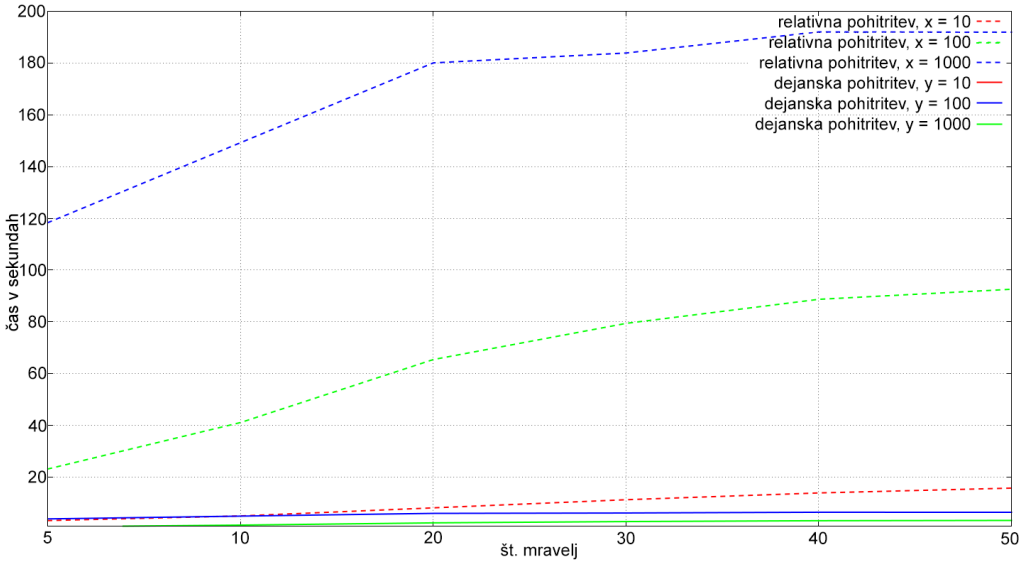
Slika 5.6: Histogram meritev časa izvajanja posameznih korakov algoritma za funkcijo sfere za parametra (tudi sekvenčno); x = 1000, y = 50.

Izračun vrednosti rešitev je pri nekaterih problemih kompleksnejši kot pri drugih. Kompleksnejši (oziroma časovno zahtevnejši) kot je izračun, tem boljše se povzporeja. Na sliki 5.6 vidimo, da se funkcija sfere in Schwefelova funkcija povzporejata slabo, ostale 4 pa (glede na redukcijo in ostale okoliščine) razmeroma dobro. Razlog za to smo posredno predstavili že v Sekciji 5.2. Sferina in Schwefelova funkcija imata zelo enostavna izračuna vrednosti rešitev, zato s povzporejanjem ne pridobimo veliko — oziroma pridobimo premalo, da bi lahko opravičili stroške povzporejanja.

5.3.4 Primerjava relativne pohitritve izvajanja celotnega algoritma z dejansko

Meritve smo opravili za 5000, 50000 in 500000 evalvacij. Na podlagi Slike (histograma) 5.6 smo se odločili, da se bomo za končni rezultat osredotočili na

Rosenbrockovo funkcijo, saj smo tam dosegli najboljšo pohitritev. Pripravili smo sledeč graf (Slika 5.7).



Slika 5.7: Graf primerjave relativne pohitritve z dejansko; a = 500000.

Črtkane črte so za relativno pohitritev, polne pa za absolutno. Rdeča barva je za problem dimenzije 10, zelena za problem dimenzije 100 in modra za problem dimenzije 1000. Graf prikazuje faktor pohitritve v odvisnosti od števila mravelj. Prva stvar, ki jo opazimo, je, da rdeče polne črte ne opazimo. Če je dimenzija problema 10 ali manj, se nam povzporejanje ne izplača. Če je problem dimenzije 100, se nam začne izplačati nekje pri 8 mravljah.

- Naš doseženi faktor relativne pohitritve za izbran problem dimenzije 1000, s 50 mravljami, po 500000 evalvacijah je: 191,944
- Naš doseženi faktor dejanske pohitritve za izbran problem dimenzije 1000, s 50 mravljami, po 500000 evalvacijah je: 6,3549

5.3.4.1 Primerjava Nvidijine platforme z AMD-jevo

Nekaj vzorčnih meritev smo opravili tudi na grafični kartici AMD Radeon HD 5870, z 1GB RAM-a. Zakaj samo nekaj vzorčnih? Omejeval nas je naš ščepec za računanje vrednosti rešitev. Naša Nvidijina grafična kartica

podpira delovne skupine velikosti do 1024 niti (in se nam lepo poklopi z dimenzijo problema, ki je lahko v našem primeru največ 1000), AMD-jeva, ki smo jo imeli na razpolago, pa samo do 256. Za meritve, ki smo jih opravili in ki smo jih želeli imeti za referenco, smo tako izbrali parametre x=250, y=50, a = 500000, $n = x \cdot y$.

- Naš doseženi faktor dejanske pohitritve na grafični kartici GeForce GT 640 za problem dimenzije 250, s 50 mravljami, po 500000 evalvacijah je: 4.9113330
- Naš doseženi faktor dejanske pohitritve na grafični kartici Radeon HD 5870 za problem dimenzije 250, s 50 mravljami, po 500000 evalvacijah je: 15.1902182687

Uporaba druge grafične kartice ima lahko torej zelo velik vpliv na čas izvajanja. Priznati moramo, da smo presenečeni nad razliko. Pripisati jo gre (med drugim tudi) različnima platformama. OpenCL je s strani AMD-ja bistveno boljše podprt kot s strani Nvidie, ki je še vedno primarno usmerjena v (svojo) CUDO.

Poglavje 6

Zaključek

V prvem, teoretičnem delu diplomske naloge smo predelali biološko navdihnjene algoritme. Naredili smo en, razmeroma površinski pregled s primeri. Biološko navdihnjeni algoritmi so danes tako široka in hitro rastoča družina algoritmov, da bi lahko naštevali in naštevali, pa še vedno ne bi našteli vseh, zato smo bili nekje primorani potegniti črto. Namen tega pregleda je bil napraviti nekakšen uvod v konkretnejše teme. Spoznali smo nekaj osnovnih pojmov in iz podpoglavja v podpoglavje tlakovali pot do za nas najbolj relevantnega algoritma, DASA.

Nato smo se od algoritmov nekoliko oddaljili in se lotili splošnonamenskega programiranja grafičnih procesorjev.

V praktičnem delu smo napisali vsega skupaj 13 ščepcev kode in na gostitelju pripravili vse potrebno za njihov zagon. To nam je bilo v izziv zlasti zato, ker smo morali prebiti jezikovno pregrado in se prilagoditi svojim omejitvam. Če bi kodo pisali neposredno za izvajanje na grafični kartici (namesto da smo temu namenu prilagajali kodo, ki mu sicer ni namenjena), bi tako koda, kot naš postopek implementacije nedvomno izgledala bistveno drugače. Kot je značilno za GPGPU, je postopek razhroščevanja dokaj mukotrpen in dolgotrajen in verjetno nam je prav ta del vzel največ časa.

Rezultati se nam zdijo zadovoljivi. Še vedno smo presenečeni nad dej-

stvom, da je taka razlika med računanjem na eni in računanjem na drugi grafični kartici. Faktor pohitritve z Nvidijine grafične kartice je slab, tisti z AMD-jeve pa povsem soliden, če vzamemo v obzir, da so optimumi niti za različne ščepce kode različni in da daleč od tega, da bi se vsi izvajali s številom niti, enakim produktu števila mravelj in dimenzije problema. Vsled tega (vsaj malo) obžalujemo, da nismo glavnine meritev izvedli na boljši grafični kartici (in da smo jo, bolj kot ne, uporabili zgolj za referenco) — je pa res, da to nima pretiranega vpliva na zgovornost rezultatov, sploh, kar se tiče relativne pohitritve in primerjave časov izvajanja posameznih korakov.

Prostor za izboljšave je. Za začetek bi lahko že večkrat omenjeni ščepec za izračun vrednosti rešitev popravili tako, da bi se lahko izvajal na vseh napravah (ki podpirajo OpenCL). Ravno tako bi lahko nekoliko koreniteje posegli v delovanje algoritma samega in le to še bolj prilagodili arhitekturi grafične kartice. Faktor pohitritve bi gotovo še izboljšali, če bi pozabili na svoje javansko okolje in bi algoritem od začetka do konca izvedli na grafični kartici. Nekatere reference, še posebej [19], predstavljajo dobra izhodišča za nadaljnji razvoj.

Kljub temu, da smo se v tem diplomskem delu močno oprli na algoritem DASA, menimo, da so principi, ki smo se jih poslužili med postopkom povzporejanja, dovolj splošni, da jih lahko uporabimo tudi pri (vsaj nekaterih) drugih biološko navdihnjenih algoritmih. V Poglavju 2 smo jih nekaj našteli, opisali njihovo delovanje, in kot vidimo, neka analognost med posameznimi algoritmi obstaja. Če ne globalno, pa vsaj znotraj posameznih polj navdiha.

Literatura

[1] Ackley Function, dostopno na:
http://www.sfu.ca/ ssurjano/ackley.html

[2] Algorithm, dostopno na:
http://www.merriam-webster.com/dictionary/algorithm

[3] B. Anthony, O. Michael. *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2006.

[4] Aparapi, dostopno na:
https://code.google.com/p/aparapi/

[5] S. Binitha, S. Siva Sathya. "A Survey of Bio inspired Optimization Algorithms", v zborniku: International Journal of Soft Computing and Engineering, 2012, str. 138-145.

[6] CUDA, dostopno na:
http://www.nvidia.com/object/cuda_home_new.html

[7] M. Dorigo. Ant Colony Optimization, dostopno na:
http://www.scholarpedia.org/article/Ant_colony_optimization

[8] A. E. Eiben, J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2007, str. 15.

[9] Evolutionary Algorithm, dostopno na:
http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/part2/faq-doc-1.html

[10] Evolutionary Design of Antennas, dostopno na:
http://idesign.ucsc.edu/projects/evo_antenna.html

[11] Griewank Function, dostopno na:
http://www.sfu.ca/ ssurjano/griewank.html

[12] Java Modulo vs Bitmask benchmarks, dostopno na:
http://chriskirk.blogspot.com/2012/05/java-modulo-vs-bitmask-benchmarks.html

[13] Jocl, dostopno na:
http://www.jocl.org/

[14] G. Kalogeropoulos, G. Ch. Sirakoulis, I. Karafyllidis. "FPGA Implementation of a Bioinspired Model for Real-Time Traffic Signals Control", Democritus University of Thrace, 2013.

[15] P. Korošec, J. Šilc, B. Filipič. *The differential ant-stigmergy algorithm.* Information Sciences, 2012.

[16] Lightweight Java Game Library, dostopno na:
http://www.lwjgl.org/

[17] Mersenne Twister, dostopno na:
http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html

[18] Meta Heuristic, dostopno na:
http://don.fed.wiki.org/view/meta-heuristic

[19] MIMD on GPU, dostopno na:
http://research.microsoft.com/apps/video/default.aspx?id=156492&r=1

[20] M. Mitchell. *An Introduction to Genetic Algorithms* . A Bradford Book, The MIT Press, 1996.

[21] Native Libraries For Java, dostopno na: https://code.google.com/p/nativelibs4java/

[22] F. Neumann, C. Witt. *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity*. Springer, 2013, preface.

[23] OpenCL, dostopno na: https://www.khronos.org/opencl/

[24] S. K. Park, K. W. Miller. "Random number generators: Good ones are hard to find", v zborniku: Computing Practices, 1988.

[25] Rastrigin Function, dostopno na: http://www.sfu.ca/ ssurjano/rastr.html

[26] Rosenbrock Function, dostopno na: http://www.sfu.ca/ ssurjano/rosen.html

[27] Schwefel Function, dostopno na: http://www.sfu.ca/ ssurjano/schwef.html

[28] Sphere Function, dostopno na: http://www.sfu.ca/ ssurjano/spheref.html

[29] Stochastic Optimization, dostopno na: http://mathworld.wolfram.com/StochasticOptimization.html

[30] B. Xing, W. Gao. *Innovative Computational Intelligence: A Rough Guide to 134 Clever Algorithms*. Springer, 2014, str. 271–272.